HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Frederic Ringsleben

Untersuchung programmierbarer Logik zur Unterstützung von Userspace-Programmen

2013

Fakultät **Mathematik/Naturwissenschaften/Informatik**

BACHELORARBEIT

Untersuchung programmierbarer Logik zur Unterstützung von Userspace-Programmen

Autor:
Frederic Ringsleben

Studiengang:
Informatik

Seminargruppe:
IF10w1-B

Erstprüfer:
Prof. Dr.-Ing. Thomas Beierlein

Zweitprüfer:
Dipl.-Ing. Fh. Thomas Oehme

Mittweida, September 2013

Bibliografische Angaben

Ringsleben, Frederic: Untersuchung programmierbarer Logik zur Unterstützung von Userspace-Programmen, 43 Seiten, 18 Abbildungen, 2 Tabellen, Hochschule Mittweida (FH), Fakultät Mathematik/Naturwissenschaften/Informatik

Bachelorarbeit, 2013

Satz: LaTeX

Referat

Die vorliegende Arbeit beschäftigt sich mit programmierbarer Logik als Ressource unter einem Betriebssystem. Es wird dokumentiert, wie die Ressource für Anwendungsprogramme zur Verfügung gestellt werden kann. Für das Bereitstellen der Ressource wird darauf eingegangen, wie sie aufgeteilt und verwaltet werden kann. Ferner wird belegt, mit welchen Möglichkeiten das Anwendungsprogramm auf ein Hardwaremodul zugreifen kann und wie eine gegenseitige Beeinflussung zwischen Hardwaremodulen und Anwendungsprogrammen verhindert wird.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Tabellenverzeichnis

IV. Abkürzungsverzeichnis

AXI	Advanced eXtensible Interface
DMA	Direct Memory Access
FPU	Floating Point Unit
HLS	High-Level Synthesis
ISR	Interrupt Service Routine
MMU	Memory Management Unit
PL	programmierbare Logik
PR	Partial Reconfiguration
PS	Prozessorsystem
SoC	System-on-a-Chip
UIO	Userspace IO

1 Einleitung

„Xilinx kauft Triscend“ [7] war die Überschrift eines Artikels von *heise online* im Jahr 2004. Das damalige Unternehmen Triscend beschäftigte sich mit Mikrocontrollern, welche über programmierbare Logik (PL) verfügten. 2011 kündigte Xilinx ein solches Design auf dem Stand der Technik an, mit dem Namen Zynq. Die Möglichkeiten und Einsetzbarkeit des Zynq-7000 scheinen grenzenlos, angefangen beim Prozessordesign bis zur programmierbaren Logik. Das Prozessordesign basiert auf dem Cortex-A9, welches in vielen heutigen Geräten Einsatz findet. Desto vielfältiger sind die Einsatzmöglichkeiten durch die PL. Controller können untergebracht werden, welche mit der Technologie mitwachsen. Wenn zum Beispiel eine Spezifikation einer Schnittstelle oder Protokolls eines Controllers nicht mehr dem Stand der Technik entspricht, könnte ein Controller ersetzt werden. Es würde ein einfaches Update genügen und es müsste nicht der komplette Chip ausgetauscht werden, was bei den meisten derzeitigen Geräten von Prinzip aus nicht funktioniert.

Der Ansatzpunkt dieser Arbeit ist ein wenig abstrakter. In heutigen Prozessoren werden viele Mechanismen verwendet, um die Bearbeitung von Problemen sowie die Verarbeitung von Daten zu beschleunigen oder zu vereinfachen. Einige dieser Mechanismen sind die Floating Point Unit (FPU) für schnelle Berechnungen, der Cache für schnelle Datenzugriffe oder die Memory Management Unit (MMU) für einen abstrakten Zugriff auf den Arbeitsspeicher. Diese Mechanismen tragen an kleinen Stellen eines Programms zur Beschleunigung bei. Durch eine PL kann ein komplexer Algorithmus von der Software- auf die Hardwareseite verlagert werden und eine Beschleunigung des gesamten Algorithmus erreicht werden. Bei dieser Problematik setzt diese Arbeit an.

Diese Arbeit soll zeigen wie Userspace-Programme mit PL interagieren können. Das soll heißen, wie mit einem Anwendungsprogramm aus dem Userspace eines Betriebssystems die Ressource PL genutzt wird. Dabei wurde das Anwendungsprogramm auf zwei Arten der Ausführung betrachtet, als alleiniges Programm und als Userspace-Programm unter einem Betriebssystem. Die Ausführung als alleiniges Programm soll aufzeigen, wie der Zugriff auf das Hardwaremodul auf vereinfachte Weise zu realisieren ist. Die eigentliche Interaktion zwischen Userspace-Programm und Hardwaremodul wird auf dem Betriebssystem Linux dargestellt. Linux wurde eingesetzt, bedingt durch die Prozessorarchitektur, welche beim Zynq auf einer ARM-Architektur basiert.

Für die Untersuchungen wurde das ZedBoard™ [3] von Digilent, Xilinx und Avnet in

PlanAhead

VIVADO

Abbildung 1.1: ZedBoard™ [3] und Xilinx Tools

Revision C als Hardware verwendet. Zum Generieren der Hardwaremodule wurden die Tools *PlanAhead 14.5*, *Vivado 2013.2* und *Vivado HLS 2013.2 [14]* eingesetzt. Das Tool *Vivado HLS 2013.2* wurde verwendet um Hardwaremodule, in Form von Netzlisten, aus C-Code zu generieren. Die von *Vivado HLS 2013.2* zur Verfügung gestellten Bibliotheken und Direktiven erweitern die Sprache C, um sinnvoll Hardwaremodule erstellen zu können. Die vorangegangene Praktikumsarbeit mit dem Thema „High Level Synthesis“ [10] beschäftigte sich mit dem Erstellen jener Hardwaremodule durch Vivado HLS. Es werden keine konkrete Beispiele dieser Software aufgezeigt, da ein abstraktes Modell ausreichend ist. Dieses Modell eines Hardwaremoduls entspricht dem EVA-Prinzip. Die Tools *PlanAhead 14.5* und *Vivado 2013.2* werden verwendet, um aus den Netzlisten und vorgefertigten Modulen einen Bitstream zu erstellen. Der Begriff Bitstream bedeutet in diesem Zusammenhang Daten, welche die Konfiguration der programmierbaren Logik für eine bestimmte Schaltung vorhält.

1.1 Problemstellung

Heutige Prozessoren arbeiten nach dem Von-Neumann-Zyklus. Diese Arbeitsweise ist für Aufgaben ungünstig, die parallel verarbeitet werden könnten. Mit programmierbarer Logik ist Parallelverarbeitung realisierbar. Der System-on-a-Chip (SoC) namens Zynq verbindet Prozessor und programmierbare Logik. Durch diesen Aufbau kann der Prozessor Aufgaben zur parallelen Verarbeitung an die programmierbare Logik des Zynq weitergeben.

Basis dieser Arbeit bildet die Frage: „Wie kann programmierbare Logik zur Unterstützung von Userspace-Programmen realisiert werden?“ Die Frage soll darauf abzielen, wie eine programmierbare Logik als Ressource unter einem Betriebssystem für Userspace-

Programme verfügbar gemacht werden kann. Es ist zu klären wie die Schnittstellen zwischen Prozessorsystem und programmierbarer Logik, Userspace-Programm und konfiguriertem Hardwaremodul, sowie Schnittstelle zur Modulkonfiguration zu realisieren sind. Unter der Schnittstelle zwischen Prozessorsystem und programmierbarer Logik ist die Anbindung der programmierbaren Logik an das Prozessorsystem auf Hardwareebene zu verstehen. Die Schnittstelle Userspace-Programm und konfiguriertem Hardwaremodul soll beschreiben, wie ein Userspace-Programm auf sein Hardwaremodul unter dem Betriebssystem zugreift. Von Xilinx wird eine Schnittstelle für die Modulkonfiguration zu Verfügung gestellt, diese lässt sich in Hinblick auf Zugriffsschutz nicht mit dieser Arbeit vereinbaren. Unter dem Bezugspunkt Zugriffsschutz sollte nicht nur die Modulkonfiguration, sondern ebenfalls die Schnittstelle Userspace-Programm und Hardwaremodul betrachtet werden. Beim Zugriffsschutz ist zu klären, wie eine gegenseitige Einflussnahme der Userspace-Programme verhindert werden kann. Die Abbildung 1.2 verdeutlicht den angestrebten Aufbau des Systems.

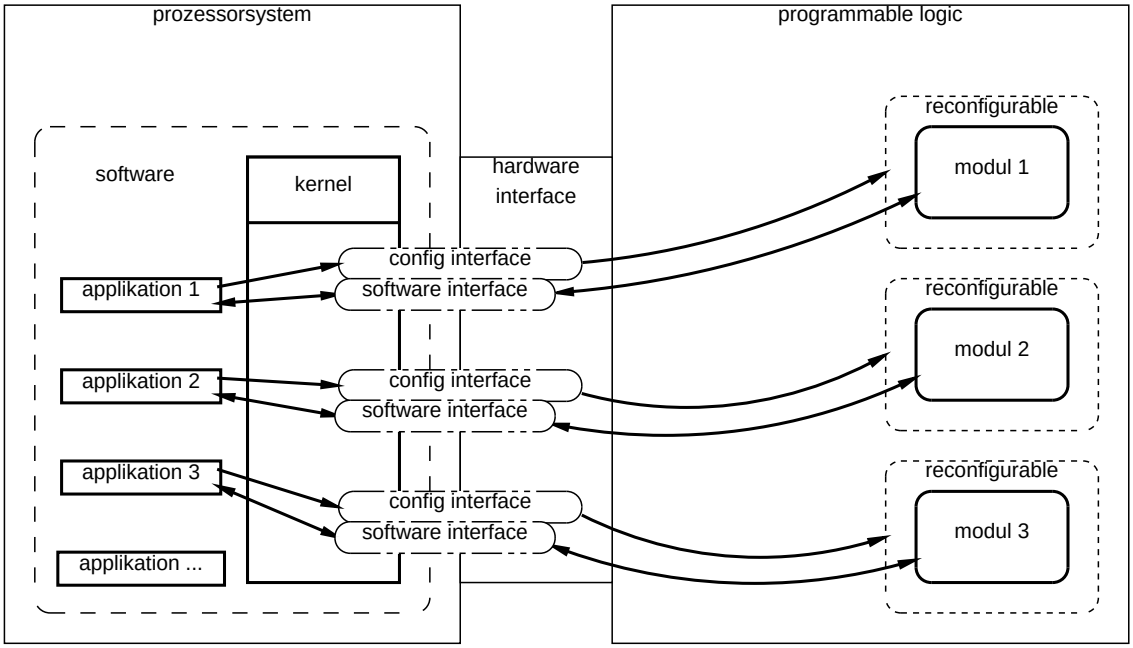

Abbildung 1.2: Prozessorsystem und programmierbare Logik

1.2 Überblick

In Kapitel 2 wird auf die Implementierung einer abstrakten Funktion als Hardwaremodul eingegangen. Im Mittelpunkt steht, wie die Daten und Signale zwischen Hardwaremodul und Prozessorsystem (PS) übertragen werden. Dafür wird auf die Techniken Memory Mapped I/O, Interrupt und Direct Memory Access eingegangen. Signale sowie Daten können über die Technik Memory Mapped I/O mit dem Modul ausgetauscht

werden. Der Interrupt dient ausschließlich der Signalisierung einer Zustandsänderung des Moduls, auf welche der Prozessor reagieren soll. Diese zwei Techniken haben den Nachteil, dass der Datenaustausch oder die Interrupt Service Routine den Prozessor in Anspruch nimmt. Im Gegensatz zu Memory Mapped I/O und Interrupt überträgt Direct Memory Access ausschließlich Daten bidirektional zwischen Speicher und Modul, ohne den Prozessor aktiv zu benutzen.

In Kapitel 3 wird auf die Anbindung der Applikation an das Hardwaremodul aus softwaretechnischer Sicht eingegangen. Zu unterscheiden ist, ob ein Programm auf einem Prozessorsystem ausgeführt wird oder mehrere Programme unter einem Betriebssystem. Wenn ein Programm ausgeführt wird, hat die Anwendung vollen Zugriff auf sämtliche Ressourcen, was in Abschnitt 3.1 dargelegt wird. Da eine Standalone-Anwendung kein Userspace-Programm im eigentlichen Sinne ist, wird hier die Ansteuerung des Hardwaremoduls aufgezeigt. Für Anwendungen unter einem Betriebssystem werden Treiber benötigt. Sogenannte Treiber sind nötig, da ein direkter Zugriff der Anwendung auf die Ressourcen aus sicherheitstechnischen Gründen Probleme bereitet. Sie sind ein Teil des Kernels und können auf die Hardware zugreifen. Mit den sogenannten Systemcalls können Anwendungsprogramme aus dem Userspace die Funktionen der Treiber nutzen. Treiber sind das Glied zwischen Applikation und Hardware. Es gibt eine weitere Möglichkeit auf die Hardware aus dem Userspace zuzugreifen. Wie Anwendungsprogramme unter dem Betriebssystem Linux die verschiedenen Ressourcen des Hardwaremoduls nutzen können, wird im Abschnitt 3.2 (S. 12 - 28) dargestellt.

Kapitel 4 zeigt Ansätze auf, wie eine Ressourcenverwaltung realisiert werden kann. Eingegangen wird darauf wie die Ressource „programmierbare Logik“ aufgeteilt werden kann. Des Weiteren wird erklärt, wie verhindert wird, dass sich Applikationen gegenseitig stören, wenn sie Hardwaremodule einsetzen wollen.

2 Grundlagen Hardwaremodul

Wie in der Einleitung erwähnt, werden hier drei Techniken Memory Mapped I/O, Interrupt und Direct Memory Access auf Hardwareebene zwischen Modul und Prozessor gezeigt. Es soll dargelegt werden, wie Informationen und Signale zwischen Hardwaremodul und Prozessorsystem übermittelt werden. Es gibt weitere Arten der Ansteuerung von Hardwaremodulen zum Beispiel über I/O-Ports oder Peripheriebus, auf welche an dieser Stelle nicht weiter eingegangen wird, da diese abhängig von der Prozessorarchitektur sind. Sämtliche Beispiele werden auf der Basis des Zynq-7000 und einem fiktiven Hardwaremodul, welches über HLS erstellt werden könnte, dargestellt. Die Funktionalität des Hardwaremoduls wurde außer Acht gelassen. Theoretisch könnte jede C-Funktion einer Applikation über HLS auf ein Hardwaremodul ausgelagert werden.

2.1 Memory Mapped I/O über AXI Bus Interface

Um Daten oder Signale von Prozessor und Modul bidirektional zu übertragen, kann der Advanced eXtensible Interface Bus (AXI) genutzt werden. Dieser Bus wird durch die „AMBA® AXI™ and ACE™ Protocol Specification“ [2] spezifiziert. Ein Modul, welches über das *AXI bus interface* mit dem Prozessor verbunden ist, überträgt über diesen Bus die Daten sowie Steuersignale. Takt sowie Reset werden über eine vom *AXI bus interface* getrennte Leitung übertragen, wie die nachfolgende Abbildung zeigt.

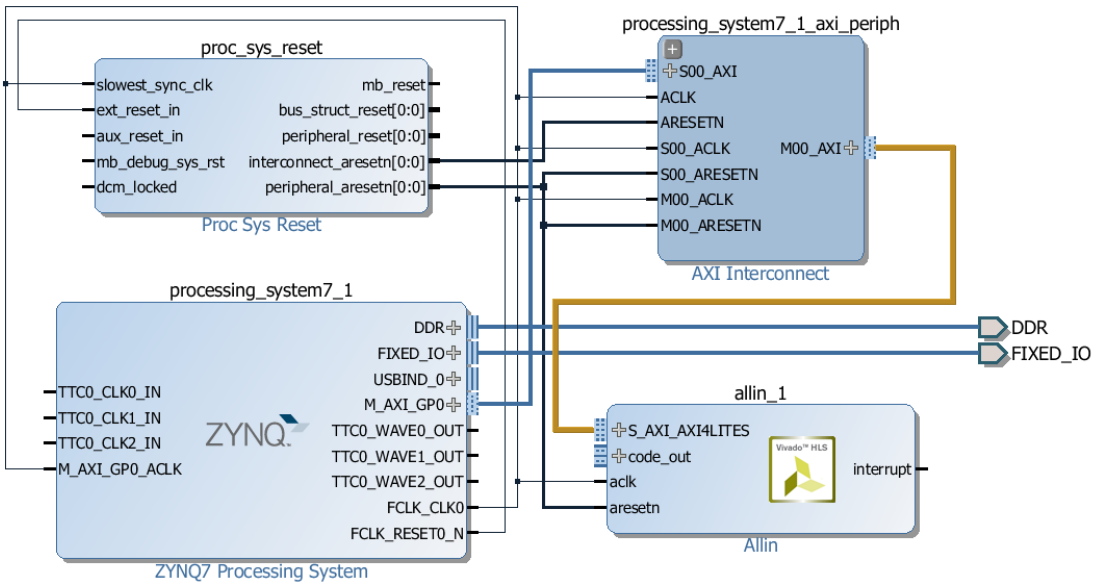

Abbildung 2.1: Hardwaremodul, AXI bus interface

Wenn ein Modul auf diese Weise an dem Prozessor angebunden ist, können die Register des Hardwaremoduls über dessen Adresse angesteuert und Daten sowie Signale geschrieben bzw. gelesen werden. Diese Register werden für die Steuerung des Moduls und für die zu verarbeitenden Daten des Moduls benötigt. Die Steuerung des Moduls wird über die untersten 16 Byte realisiert, welche vom Modul im Adressraum eingeblendet werden. Ein Beispiel, wie die Steuerregister in den Adressraum eingeblendet werden, stellt die folgende Liste dar.

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
```

Wenn Daten über das *AXI bus interface* übertragen werden sollen, blendet das Hardwaremodul weitere Datenregister im Adressraum oberhalb der 16 Byte ein. Wie ein Modul mit *AXI bus interface* von der Applikation genutzt werden kann, wird in Kapitel 3 ab Seite 11 aufgezeigt.

2.2 Interrupt

Nach Jonathan Corbet, Alessandro Rubini und Greg Kroah-Hartman wird ein Interrupt wie folgend definiert: „An Interrupt is simply a signal that the hardware can send when it wants the processor´s attention“ [5, S. 258]. Demzufolge signalisiert der Interrupt dem Prozessorsystem verschiedene Zustände der Hardware. Per Interrupt kann ein Modul das erfolgreiche Beenden einer Aufgabenstellung an den Prozessor signalisieren. In diesem Fall wird der aktuell laufende Prozess unterbrochen und eine Interrupt-Service-Routine (ISR) ausgeführt. Die ISR wird im Abschnitt 3.2.1 detaillierter beschrieben. Die

Abbildung 2.2 zeigt die direkte Verbindung der Interrupt-Leitung zwischen Modul und Prozessor.

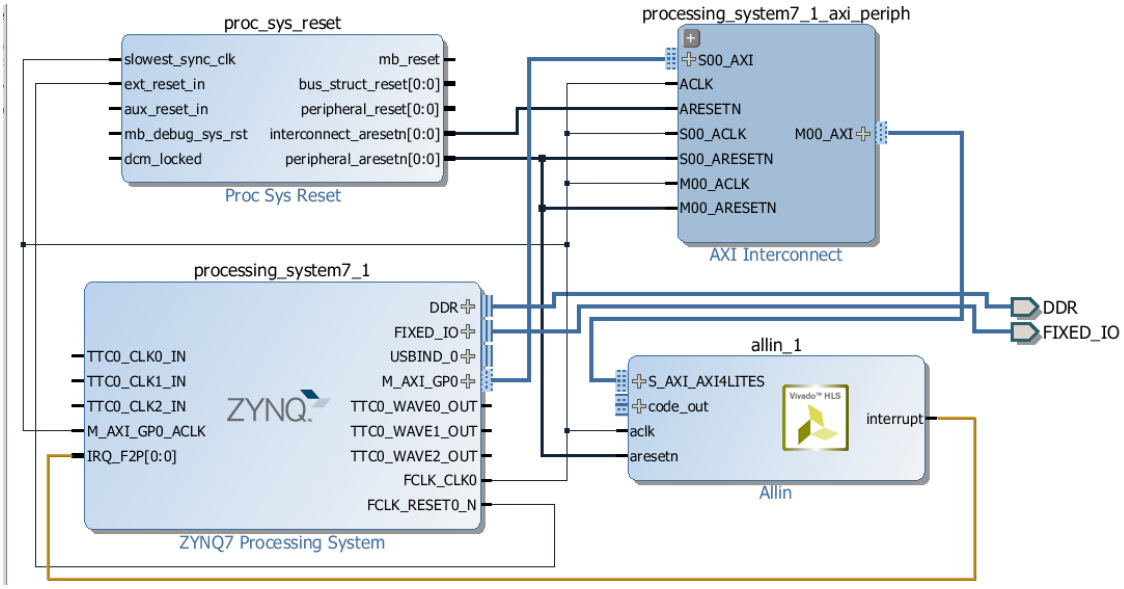

Abbildung 2.2: Hardwaremodul Interrupt

Module mit Interrupt werden generiert, wenn einer Funktion bei Vivado HLS als Directive nicht *ap_ctrl_none* zugeordnet wurde. Directiven beschreiben in Vivado HLS das Handshake und Interface eines Moduls. Um Interrupts nutzen zu können, müssen diese über die Steuerregister aktiviert werden. Interrupt-Sharing wird durch die Steuerregister gewährleistet. Durch dieses Interrupt-Sharing können mehrere Module einen Interrupt nutzen, wenn über die Steuerregister der Module bestimmbar ist, welches den Interrupt ausgelöst hat. Im Abschnitt 3.2.1 wird im Detail auf die Implementierung seitens der Software eingegangen.

2.3 Direct Memory Access mit AXI Stream Bus Interface

Direct Memory Access (DMA) ist ein Verfahren, bei dem ausschließlich Daten zwischen Modul und Speicher übertragen werden. Das Besondere bei diesem Verfahren ist, dass der Prozessor nicht aktiv mit der Datenübertragung oder dem Ausführen von Routinen, zum Beispiel der Interrupt Service Routine, beschäftigt wird. Das Übertragen der Daten übernimmt ein DMA-Controller. Während der Datenübertragung können weitere Prozesse auf dem Prozessor ausgeführt werden. Damit der DMA-Controller seine Arbeit verrichten kann, muss der Prozessor den Zugriff auf den Systembus als Bus-Master an den DMA-Controller abgeben. Abbildung 2.3 zeigt den prinzipiellen Aufbau.

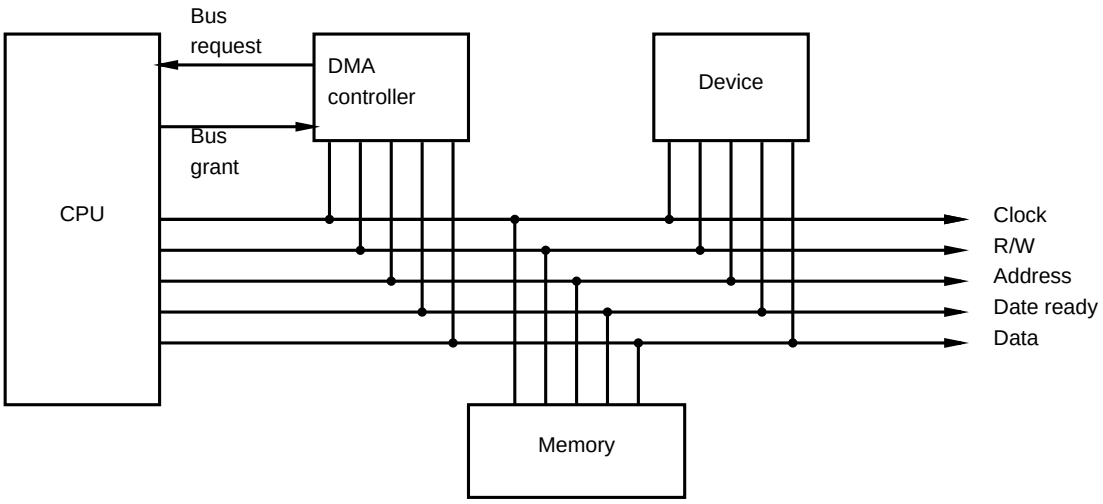

Abbildung 2.3: A bus with a DMA controller. [11, S. 186]

Die prinzipielle Anbindung eines DMA-Controllers in dieser Abbildung weist Unterschiede zur „AMBA® AXI™ and ACE™ Protocol Specification“ [2, S. A1-20, A7-90] auf. Einer dieser Unterschiede betrifft den Systembus. In der Spezifikation wird darauf verwiesen, dass zwei separate Kanäle zum Lesen und Schreiben zur Verfügung stehen. Daten können zeitgleich auf dem Bus gelesen und geschrieben werden, im Unterschied zur Abbildung 2.3, in welcher das adressierte Register bzw. die adressierte Speicherzelle beschrieben oder gelesen werden kann.

Um ein Modul über DMA anzubinden, gibt es mehrere Möglichkeiten, von welchen hier eine beschrieben wird. Der in Vivado enthaltene *IP-Core „AXI Direct Memory Access“ [13]* wird als DMA-Controller eingesetzt. Dieser *IP-Core* verfügt über zwei Systembusschnittstellen und ein *AXI Stream Bus Interface*. Die Systembusschnittstellen dienen dem Ansteuern des DMA-Controllers durch den Prozessor und dem Benutzen des Systembusses durch den DMA-Controller. Das Hardwaremodul wird über das *AXI Stream Bus Interface* mit dem Controller verbunden. Bei dem *AXI Stream Bus Interface* werden die Daten mittels FIFO-Verfahren übertragen. Die Daten müssen im Modul entgegengenommen und den entsprechenden Registern zugeordnet werden, bzw. vom Modul in richtiger Reihenfolge aus den Registern über den Bus übertragen werden. Wichtig bei diesem Verfahren ist die Datenreihenfolge, welche bei Entwicklung des Moduls und des zugehörigen Treibers synchron zu Realisieren ist.

Zur Ansteuerung des Xilinx DMA-Controllers werden seine Steuerregister [13, S. 13ff.] im Adressbereich eingeblendet. Über diese Adressen der Register können entsprechende Informationen zur Steuerung des Controllers geschrieben und gelesen werden. Des Weiteren befindet sich eine Interrupt-Leitung am *IP-Core*, welche mit dem Prozessor verbunden werden kann. Abbildung 2.4 zeigt wie Modul, DMA-Controller, Speicher

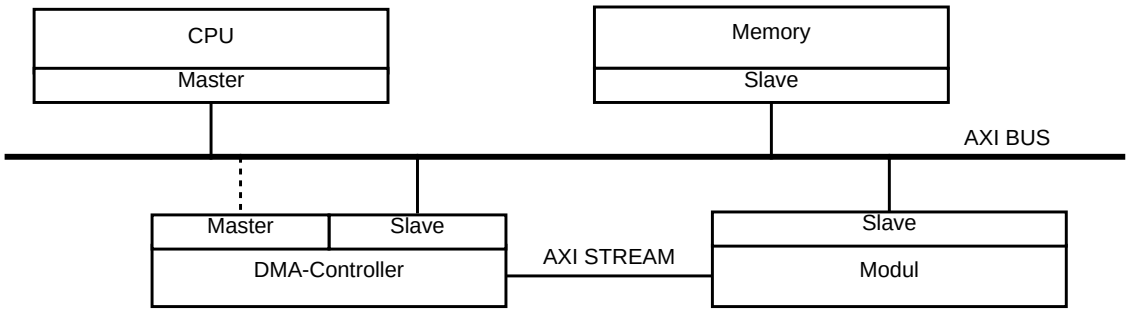

Abbildung 2.4: DMA-Controller mit *AXI Stream Bus Interface*

und CPU über die *AXI interfaces* verbunden sind. Durch diese Verbindungen sind folgende Zugriffe realisierbar:

- Der Prozessor als Busmaster greift über die jeweiligen Adressen auf den Speicher oder die Register von Modul sowie DMA-Controller zu.
- Die Daten können über das *AXI bus interface* zwischen Speicher und DMA-Controller ausgetauscht werden. Gleichzeitig werden diese Daten zwischen Modul und DMA-Controller über das *AXI Stream Bus Interface* transferiert.

Erteilt der Prozessor dem DMA-Controller einen Auftrag, kann der DMA-Controller beim Prozessor die Benutzung des Systembusses anfordern. Gibt der Prozessor den Systembus für den DMA-Controller frei, ist der DMA-Controller Busmaster und kann auf den Speicher zugreifen. Der Prozessor erhält seinen Zugriff auf dem Systembus wieder, wenn er den Bus benötigt oder die Übertragung des DMA-Controllers beendet wurde.

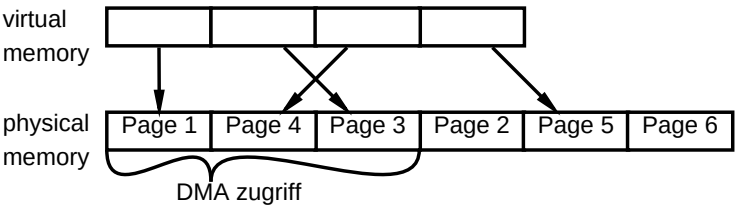

Abbildung 2.5: virtuelle physischer Speicher

Benutzt man den DMA-Controller im Modus *Simple DMA Mode,* werden dem DMA-Controller Startadresse und Länge der Daten mitgeteilt, welche geschrieben oder gelesen werden sollen. Wichtig ist, das die zu übertragende Daten in aufeinanderfolgenden Pages liegen müssen. Das ist vor allem für die Softwareimplementierung wichtig, weil von Userspace-Programmen angeforderter Speicher bis auf Ausnahmen nicht physischer Speicher, sondern virtueller Speicher ist. Virtueller Speicher muss nicht zwangsläufig aus zusammenhängenden physischen Pages bestehen. Wenn die Länge der zu übertragenden Daten länger als eine Page ist und die zweite physische Page nicht hin-

ter der Ersten liegt, werden falsche Daten übertragen. Wie eine Datenkommunikation zwischen Applikationen und Modul über DMA realisierbar ist, wird im Abschnitt 3.2.2 aufgezeigt.

3 Grundlagen Software

3.1 Standalone, Programm ohne Betriebssystem

Als Standalone wird eine Anwendung bezeichnet, welche ohne übergeordnetes Betriebssystem ausgeführt wird. Hier kann die Anwendung direkt auf sämtliche Ressourcen zugreifen. Diese Tatsache lässt den Programmieraufwand für eine Standalone-Anwendung gering werden. Es kann direkt auf die Register eines Moduls zugegriffen werden. Eine Standalone-Anwendung ist kein Userspace-Programm im eigentlichen Sinne und soll zum Einstieg zeigen, wie die Kommunikation zwischen Hardware und Software realisierbar ist. In folgendem Programmlisting wird der Zugriff auf ein abstraktes Modul per Memory Mapped I/O gezeigt.

```
#include <asm/io.h>

#define HARDWARE_ADDR    ...
#define HARDWARE_IN      ...
#define HARDWARE_OUT     ...

int call_hardware_function(int[] data_in, int length_in,
        int[] data_out, int length_out) {
    int ret, timeout = 0;

    // Datentransport vom Speicher zum Hardwaremodul
    memcpy(HARDWARE_ADDR + HARDWARE_IN, data_in, length_in);

    // Verarbeitung im Hardwaremodul starten
    HARDWARE_ADDR = 1;

    // Warte auf Verarbeitungsende des Hardwaremoduls (Polling)
    do {
        ret = HARDWARE_ADDR;
        timeout++;
    } while ((!(ret & 0x02)) && timeout < 100000);

    // Timeout
    if (timeout >= 100000) {
        // Fehlerbehandlung
        return EXIT_FAILURE;
    }

    // Datentransport vom Hardwaremodul zum Speicher
```

```
    memcpy(data_out, HARDWARE_ADDR + HARDWARE_OUT, length_out);
    return EXIT_SUCCESS;
}
```

Die Arbeitsweise im Umgang mit einem Hardwaremodul wird im Quellcode sichtbar. Zuerst werden die zu verarbeitenden Daten in die Register des Moduls übertragen und anschließend die Verarbeitung der Daten im Modul gestartet. Über die *while*-Schleife wird die Beendigung des Verarbeitungsprozesses überprüft. Dieses ständige Prüfen eines eintreffenden Ereignisses wird Polling genannt und weiter unten erklärt. Wenn der Verarbeitungsprozess vor der Zeitbeschränkung beendet wurde, werden die verarbeiteten Daten zurück in den Speicher übertragen. Die Verarbeitung ist abgeschlossen.

Polling

Polling wird im Werk Taschenbuch Mikroprozessortechnik von Prof. Dr.-Ing. Thomas Beierlein und Prof. Dr.-Ing. Olaf Hagenbruch erklärt, mit den Worten: „Im Polling-Verfahren wird per Programm ständig kontrolliert, ob Daten von Eingabeeinheiten eingelesen oder an Ausgabeeinheiten ausgegeben werden können. Dazu werden in Abfrageschleifen entsprechende Statusinformationen dieser Einheiten ausgewertet.“ [4, S. 37] Dieses Verfahren besitzt Vor- und Nachteile gegenüber dem Interrupt mit Interrupt Service Routine. Es kommt auf das Anwendungsszenario an, welches Verfahren sinnvoller ist. Wie Interrupt und Interrupt Service Routine funktioniert wird in den entsprechenden Abschnitten aufgezeigt.

3.2 Zeichen-Treiber, Zugriff über Betriebssystemtreiber

In einem Linux-System wird der direkte Zugriff einer Applikation auf die physischen Registeradressen durch Speicherschutzmechanismen verhindert. Um das Hardwaremodul ansteuern zu können, benötigt man einen Treiber. Dieser kann auf die Registeradressen des Hardwaremoduls zugreifen. In Abbildung 3.1 wird vereinfacht dargestellt, wie die Applikation über einen Treiber auf das Hardwaremodul zugreifen muss, welches per Memory Mapped I/O angebunden ist.

Weil eine komplette Einführung in die Linux-Treiber zu weit führt, sei an dieser Stelle auf das Buch „Linux-Treiber entwickeln“ von Jürgen Quade und Eva-Katharina Kunst [9] hingewiesen. Um auf das Hardwaremodul zugreifen zu können, reichen im Wesentlichen

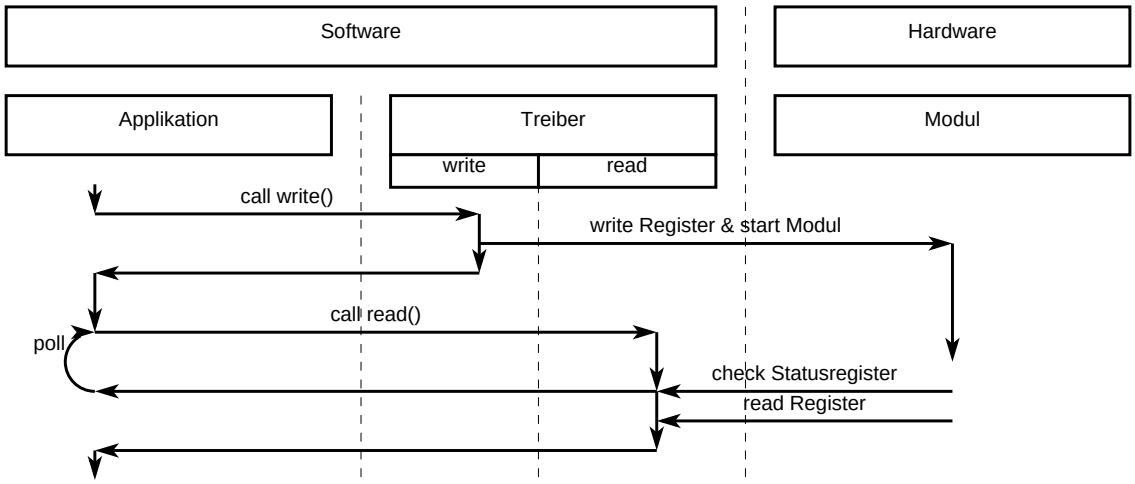

Abbildung 3.1: Schema: Linux Treiber

die vier Funktionen *open()*, *close()*, *write()* und *read()*. Darüber hinaus können weitere Funktionen implementiert werden, wie *select()*, *fcntl()*, *lseek()* und *ioctl()*. Diese werden genutzt um eine Steuerung der Hardware zu implementieren, worauf hier nicht weiter eingegangen wird. Die wesentlichen Funktionen für die Implementierung eines Treibers für den Zugriff auf das Hardwaremodul werden im Folgenden beschrieben:

open() Zu Beginn muss der Adressbereich des Hardwaremoduls beim Kernel angefordert werden. Die Funktion *request_mem_region()* übernimmt diese Aufgabe und liefert ein *struct resource** zurück. Um auf die virtuelle Speicheradresse zugreifen zu können, muss mit *ioremap()* der *struct resource** auf eine virtuelle Adresse abgebildet werden. Die zurückgegebenen Zeiger müssen für die spätere Nutzung, sowie Freigabe im Datenbereich einer Treiberinstanz hinterlegt werden. Nachfolgend ein Beispiel wie die *open()*-Funktion aussehen könnte.

```
#include <asm/io.h>
#include <linux/slab.h>
#include <linux/ioport.h>

#define HARDWARE_ADDR           ...
#define HARDWARE_ADDR_RANGE     ...

struct instant_data {
    struct resource *resource_ptr;
    void *hardware_ptr;
};

static int driver_open(struct inode *devicefile, struct file *instance) {
    struct instant_data *iptr;
```

```
    iptr = (struct instant_data*) kmalloc(sizeof (struct instant_data)
            , GFP_KERNEL);

    // Adressbereich des Hardwaremodul anfordern
    iptr->resource_ptr = request_mem_region(HARDWARE_ADDR
            , HARDWARE_ADDR_RANGE
            , "device_name");

    // physische Adresse auf virtuelle Adresse abbilden
    iptr->hardware_ptr = ioremap(HARDWARE_ADDR
            , HARDWARE_ADDR_RANGE);

    instance->private_data = (void*) iptr;
    return 0;
}
```

close() Die Funktion *close()* muss die angeforderten Ressourcen freigeben.

```
#include <asm/io.h>
#include <linux/slab.h>
#include <linux/ioport.h>

#define HARDWARE_ADDR           ...
#define HARDWARE_ADDR_RANGE     ...

static int driver_close(struct inode *devicefile, struct file *instance) {

    // Abbildung der virtuellen Adresse aufheben
    iounmap(((struct instant_data*) instance->private_data)->hardware_ptr);

    // Adressbereich des Hardwaremoduls freigeben
    release_mem_region(HARDWARE_ADDR, HARDWARE_ADDR_RANGE);

    // Speicher freigeben
    kfree(instance->private_data);
    return 0;
}
```

write() Um die Daten der Applikation in die Register des Moduls zu kopieren und das Modul zu starten, müssen diese Funktionalitäten in der Funktion *write()* implementiert werden. Die Register können unter der virtuellen Adresse aus der Funktion *open()* angesprochen werden. Die Daten können nicht direkt vom Userspace in die Register des Moduls transferiert werden. Eine Erklärung ergibt sich aus dem Memory Management [9, S. 109ff. und 126ff.]. Das macht es notwendig die Daten zweimal zu kopieren mit *copy_from_user()* vom Userspace zum Kernelspace und *mit memcpy_toio()* vom Kernelspace in die Register. Unmittelbar nach

Beendigung der Übertragung der Daten in die Register des Moduls kann die Verarbeitung des Moduls gestartet werden.

```
#include <asm/io.h>
#include <linux/slab.h>
#include <linux/ioport.h>
#include <linux/kernel.h>

#define HARDWARE_WRITE              ...
#define HARDWARE_WRITE_LENGTH       ...

ssize_t driver_write(struct file *instance, const char __user *buffer,
        size_t max_bytes_to_write, loff_t *offset) {
    void *hardware = ((struct instant_data*) instance->private_data)->hardware_ptr;
    size_t to_copy, not_copied;

    // nur für kleine Puffer, ansonst Speicher resevieren
    char kernelmem[HARDWARE_WRITE_LENGTH];

    // Datentransport von Userspace zum Kernelspace
    to_copy = min(HARDWARE_WRITE_LENGTH, max_bytes_to_write);
    not_copied = copy_from_user(kernelmem, buffer, to_copy);

    // Datentransport vom Speicher zum Hardwaremodul
    memcpy_toio(*hardware + HARDWARE_WRITE, kernelmem, to_copy);

    // Hardwaremodul starten
    *hardware = 1;
    return to_copy - not_copied;
}
```

read() Ein Ergebnis bzw. die verarbeiteten Daten werden über die Funktion *read()* abgefragt. Hier ist ein zweimaliges Kopieren der Daten aus den Registern in den Kernelspace und vom Kernelspace in den Userspace notwendig. Das zweimalige Kopieren ergibt sich aus dem selben Grund des Memory Management [9, S. 109ff. und 126ff.], wie in der Funktion *write()*. Dies wird realisiert durch die Funktionen *memcpy_fromio()* und *copy_to_user()*. Beachtet werden muss, dass das Hardwaremodul parallel zum Prozess seine Daten verarbeitet. Nicht sichergestellt werden kann, dass das Hardwaremodul seine Datenverarbeitung abgeschlossen hat, wenn die Applikation den Systemcall read() aufruft. Für dieses Problem gibt es zwei Lösungsmöglichkeiten. Entweder wird das Hardwaremodul wiederholend abgefragt, oder das Hardwaremodul signalisiert sein Verarbeitungsende dem Prozessorsystem. Das wiederholende Abfragen wurde unter dem Namen Polling vorgestellt. Das Signalisieren geschieht über den Interrupt (siehe Abschnitt 2.2) und

die Behandlung des Interrupts wird im Abschnitt Interrupt Service Routine aufgezeigt. Aufgrund das Interrupt und Interrupt Service Routine in separaten Kapiteln aufgezeigt werden, wird im folgenden Quellcode der Funktion *read()* das Polling-Verfahren verwendet. Zu sehen ist das Prüfen des Verarbeitungsendes im Quellcode. Wenn diese Prüfung fehlschlägt, wird die Funktion *read()* mit einem Fehlercode, z.B. *EAGAIN* abgebrochen. Die Applikation wertet die Rückgabe aus und reagiert dementsprechend. Erhält die Applikation den Fehlercode, muss sie das Bearbeitungsende wiederholend beim Treiber abfragen. Wichtig ist, dass der Userprozess unterbrochen werden kann, damit ein anderer Userprozess ausgeführt werden kann. Ein Polling im Kernelspace hat den Nachteil, dass es den Prozessor belegt und keine Userprozesse auf dem Prozessor ausgeführt werden können.

```
#include <asm/io.h>
#include <linux/slab.h>
#include <linux/ioport.h>
#include <linux/kernel.h>

#define HARDWARE_READ              ...
#define HARDWARE_READ_LENGTH       ...

ssize_t driver_read(struct file *instance, char __user *buffer,
        size_t max_bytes_to_read, loff_t *offset) {
    void *hardware = ((struct instant_data*) instance->private_data)->hardware_ptr;
    size_t to_copy, not_copied;

    //nur für kleine Puffer, ansonst Speicher reservieren
    char kernelmem[HARDWARE_READ_LENGTH];

    //Prüfe Hardwaremodul auf Beendigung seiner Verarbeitung
    if (!(*hardware & 0x02)) {
        return -EAGAIN;
    }

    // Datentransport von Kernelspace zum Userspace
    to_copy = min(HARDWARE_READ_LENGTH, max_bytes_to_read);
    not_copied = copy_to_user(buffer, kernelmem, to_copy);

    // Datentransport vom Hardwaremodul zum Speicher
    memcpy_fromio(kernelmem, *hardware + HARDWARE_READ, to_copy);

    return to_copy - not_copied;
}
```

3.2.1 Interrupt Service Routine

Eine Interrupt Service Routine (ISR) wird ausgeführt, wenn die aktuelle Verarbeitung des Prozessors durch einen Interrupt unterbrochen wird [9, S. 159]. Durch dieses System aus Interrupt und ISR kann ein Vorteil entstehen, wenn das Modul lange Verarbeitungszeiten besitzt. Ob ein Modul seine Arbeit abgeschlossen hat, muss nicht wie bisher ständig geprüft werden. Dies gelingt indem die Funktion *read()* prüft, ob die Bearbeitung beendet wurde und sich schlafen legt, wenn es noch nicht der Fall war. Während die Verarbeitung im Modul stattfindet, kann ein anderer Prozess auf dem Prozessor ausgeführt werden. Wenn eine ISR durch zugehörigen Interrupt ausgelöst wurde, überprüft sie die Steuerregister des Moduls auf das Fertigstellen seiner Bearbeitung. Ist diese Prüfung erfolgreich, kann die schlafende Funktion *read()* aufgeweckt werden. Nach dem Aufwecken der Funktion *read()* beginnt diese mit dem Einlesen der Daten vom Hardwaremodul.

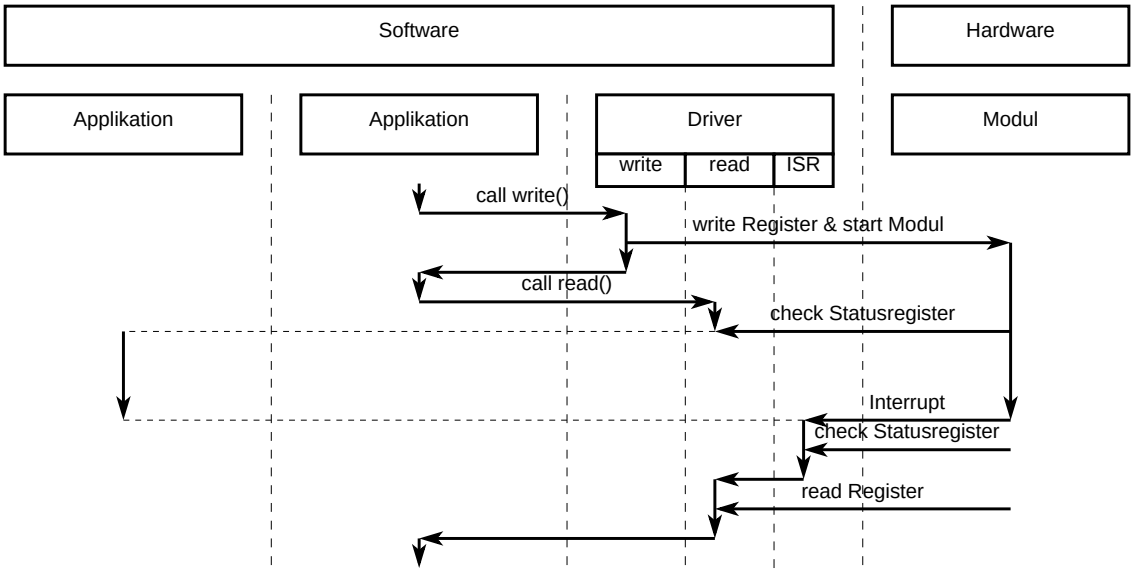

Abbildung 3.2: Schema: Linux ISR-Treiber

Das Testen eines Hardwaremoduls, ob es seine Verarbeitung beendet hat, ist aus zwei Gründen sinnvoll:

- Der Interrupt kann mehr signalisieren, als das Verarbeitungsende seines Moduls. Es kann zum Beispiel signalisiert werden, dass ein Fehler aufgetreten ist.
- Interrupt-Sharing ermöglicht, dass zwei Module den selben Interrupt nutzen können.

Der zweite Grund, Interrupt-Sharing wurde im Abschnitt 2.2 kurz erwehnt. In der ISR muss geprüft werden, welches der Hardwaremodule den Interrupt ausgelöst hat. Zum

Prüfen seiner Hardware muss für jedes Modul eine eigene ISR angemeldet sein. Wurde in der ISR des Hardwaremoduls festgestellt, dass der Interrupt von dessen Hardwaremodul ausgelöst wurde, ist die ISR mit IRQ_HANDLED zu quittieren. Wurde der Interrupt nicht vom eigenen Hardwaremodul ausgelöst, muss die ISR mit IRQ_NONE quittiert werden, damit die nächst ISR abgearbeitet werden kann. Das folgende Programmbeispiel einer ISR demonstriert die Prüfung eines Hardwaremoduls mit Interrupt-Sharing.

```
static DECLARE_WAIT_QUEUE_HEAD(read_wait_irq);

irqreturn_t driver_isr(int irq, void *dev_id) {
    void *hardware = ((struct instant_data*) instance->private_data)->hardware_ptr;
    // prüfe Modul
    if (!readb(*hardware) & 0x02) {
        // Interrupt wurde nicht durch Modul des Treibers ausgelöst
        return IRQ_NONE;
    }

    // schlafende read()-Funktion wecken
    wake_up(&read_wait_irq);

    // Interrupt als bearbeitet gegenüber dem Kernel melden
    return IRQ_HANDLED;
}
```

Weiterhin muss die ISR beim Kernel an- bzw. abgemeldet werden, wenn der Treiber geladen bzw. entladen wird. Die Funktion *read()* muss ebenfalls angepasst werden, damit der Prozess schlafen gelegt wird, wenn das Hardwaremodul seine Verarbeitung noch nicht abgeschlossen hat. Die Anpassungen für die Nutzung der ISR wird durch Tabelle 3.1 verdeutlicht.

activity	source code
register ISR in *open()*	`request_irq(HARDWARE_INT, driver_isr, IRQF_SHARED, TEMPLATE, &driver_object)`
de-register ISR in *close()*	`free_irq(HARDWARE_INT, &driver_object);`
remove from read() for wait	`if (!(ret & 0x02)) { return -EAGAIN; }`
insert into read() in place of return	`if (!readb(HARDWARE_ADDR) & 0x02) { wait_event_interruptible(read_wait_irq, readb(HARDWARE_ADDR)&0x02); }`

Tabelle 3.1: ISR-Anpassungen

3.2.2 Direct Memory Access

Direct Memory Access auf Hardwareebene wurde im Abschnitt 2.3 ausführlich erklärt. Der DMA-Controller ist eine Hardwarekomponente, die durch den Treiber gesteuert werden muss. Für die Implementation wird kein weiterer Treiber nötigt, die Steuerung des DMA-Controllers kann im Treiber für das Hardwaremodul mit erledigt werden. Abbildung 3.3 zeigt den Ablauf einer möglichen Treiberimplementierung.

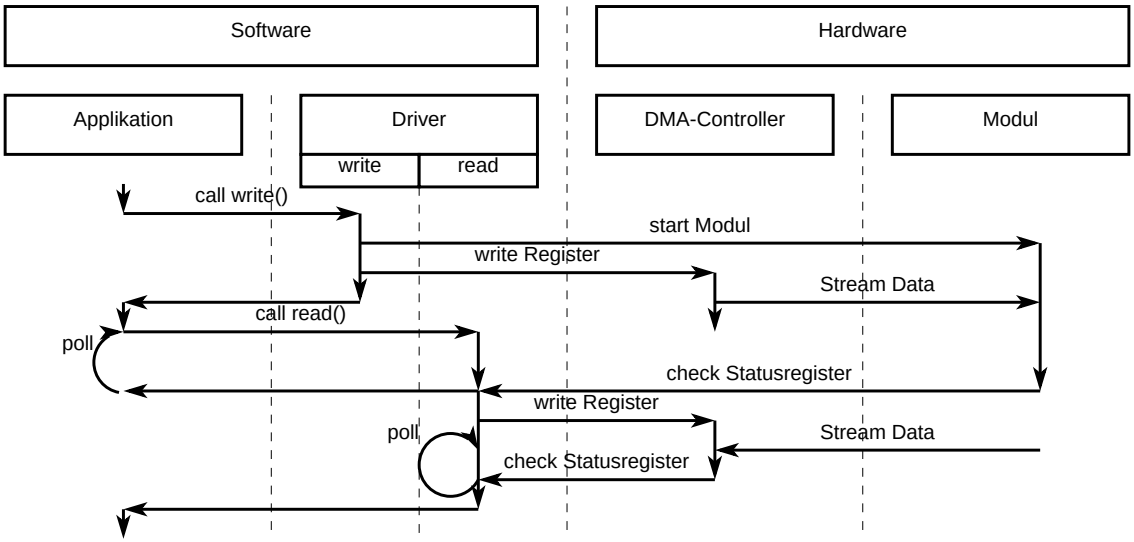

Abbildung 3.3: Schema: Liunx DMA-Treiber

In der Abbildung wird Polling aus Gründen der Übersichtlichkeit dargestellt. Zum Nachteil wird dieses Polling auf die eigene Hardware, weil die CPU während der Übertragung andere Prozesse ausführen könnte. Um diesen Nachteil zu verhindern, sollte mit Interrupts gearbeitet werden. Der aus Abbildung 3.3 verdeutlichte Ablauf wird folgend mit den nötigen Anpassungen des Treibers erläutert. Dieser unterscheidet sich in wesentlichen Punkten vom Zeichen-Treiber aus Abschnitt 3.2.

Die Funktion *write()* kann das Hardwaremodul starten, sobald die Register beschrieben wurden, weil das Hardwaremodul auf den Stream vom DMA-Controller wartet. Wenn die Datenkommunikation zwischen Software und Hardwaremodul ausschließlich über das *AXI Stream Bus Interface* geregelt wird, kann das Modul als Erstes in der Funktion *write()* über sein Kontrollregister gestartet werden. Des Weiteren kann ein Modul regelrecht ohne Handshake betrieben werden. Das bedeutet, ein Modul beginnt mit seiner Verarbeitung, sobald alle Daten im Modul eingetroffen sind und beendet seine Verarbeitung mit dem Senden der Daten über einen Stream zum DMA-Controller. Wichtig ist dieser Punkt für das partielle Rekonfigurieren, siehe Abschnitt 4.1, weil bei dieser Methode kein Adressraum für das Modul benötigt wird.

Die Funktion *write()* muss DMA-Speicher anfordern, in welchen dem DMA-Controller die Daten für das Modul hinterlegt werden. Auf ein Puffer-Array im Treiber oder eine Anforderung von Kernel-Speicher mit *kmalloc()* kann verzichtet werden, begründet durch die Anforderung von DMA-Speicher. Der DMA-Speicher nimmt die Funktion des Puffers ein. Von Vorteil bei Verwendung von DMA im Treiber ist das einmalige Kopieren von Daten. Mussten im Treiber ohne DMA die Daten mit *copy_from_user()* und *memcpy_toio()* zweimal kopiert werden, genügt *copy_from_user()* beim Treiber mit DMA. Daten können direkt aus dem virtuellen Userspace Speicher in den angeforderten DMA-Speicher kopiert werden, weil der DMA-Controller für die Übertragung der Daten aus dem DMA-Speicher zum Modul sorgt.

```
#include <asm/io.h>
#include <linux/slab.h>
#include <linux/ioport.h>
#include <linux/kernel.h>
#include <linux/dma-mapping.h>

#define HARDWARE_WRITE_LENGTH                   ...
#define DMA_CONTROLLER_WRITE_CTRL               ...
#define DMA_CONTROLLER_WRITE_STAT               ...
#define DMA_CONTROLLER_WRITE_BUFFER_ADDR        ...
#define DMA_CONTROLLER_WRITE_BUFFER_LEN         ...

static struct device *dev;
dma_addr_t *dma_handle;

ssize_t driver_write(struct file *instance, const char __user *buffer,
        size_t max_bytes_to_write, loff_t *offset) {
    void *dma_hardware = ((struct instant_data*) instance->private_data)->hardware_ptr;
    void *dma_mem;
    size_t to_copy, not_copied;

    // Hardwaremodul starten (optional)
    *hardware = 1;

    // DMA-Speicher anfordern
    dma_mem = dma_alloc_coherent(dev
            , HARDWARE_WRITE_LENGTH
            , dma_handle
            , GFP_KERNEL);
    if (dma_mem == NULL) {
        return -EAGAIN;
    }

    // Datentransport von Userspace zum DMA-Speicher
```

```
    to_copy = min(HARDWARE_WRITE_LENGTH, max_bytes_to_write);
    not_copied = copy_from_user(kernelmem, buffer, to_copy);

    // DMA-Controller starten
    writeb(*dma_hardware + DMA_CONTROLLER_WRITE_CTRL, 1);
    // DMA-Controller Adresse des DMA-Speichers übergeben
    writel(*dma_hardware + DMA_CONTROLLER_WRITE_BUFFER_ADDR, dma_mem);
    // DMA-Controller Länge des DMA-Speichers übergeben
    writel(*dma_hardware + DMA_CONTROLLER_WRITE_BUFFER_LEN
            , HARDWARE_WRITE_LENGTH);

    // Polling und Freigabe möglichst durch Interrupt und ISR ersetzen
    // Beendigung der Übertragung des DMA-Controllers abwarten
    while (!((*dma_hardware + DMA_CONTROLLER_WRITE_STAT)&0x02));
    // DMA-Speicher freigeben
    dma_free_coherent(dev, HARDWARE_WRITE_LENGTH, dma_mem, dma_handle);

    return to_copy - not_copied;
}
```

Bei der Funktion *read()* ohne Interrupt muss über ein Statusregister geprüft werden, zu welchem Zeitpunkt das Hardwaremodul seine Verarbeitung beendet hat. Darauf folgend muss die Funktion *read()* DMA-Speicher anfordern, in welchen der DMA-Controller die aus der Verarbeitung resultierenden Daten des Moduls ablegen kann. Anschließend muss dem DMA-Controller die Adresse und Größe des DMA-Speichers mitgeteilt werden. Wenn der DMA-Controller seine Übertragung der Daten in den DMA-Speicher beendet hat, signalisiert er das über seine Statusregister bzw. Interrupt. Wurde das Flag für den Erfolg der Übertragung gesetzt, kann der Treiber die Daten in den Userspace Speicher kopieren. Beim Kopieren der Daten aus dem DMA-Speicher, in den Userspace Speicher, verhält es sich wie bei der Funktion *write()* aus Punkt 2. Ein *copy_to_user()* genügt und es ist kein *memcpy_fromio()* nötig um die Daten in einen Puffer zu kopieren, weil die Daten im Kernelspace liegen.

Auch bei der Funktion *read()* kann auf ein Handshake mit dem Hardwaremodul verzichtet werden. Das bedeutet, dass ein Hardwaremodul ohne Interrupt und Statusregister auskommt, wenn es über das *AXI Stream Bus Interface* mit dem Prozessorsystem verbunden ist. Das Hardwaremodul signalisiert nicht sein Verarbeitungsende über Interrupt oder Statusregister, sondern versucht die Daten per Stream an den DMA-Controller zu übertragen. Dieser nimmt die Daten entgegen und leitet sie an den Speicher weiter, wenn ihm die Adresse und Größe eines passenden DMA-Speichers über den Treiber mitgeteilt wurde.

```
#include <asm/io.h>
#include <linux/slab.h>
#include <linux/ioport.h>
#include <linux/kernel.h>
#include <linux/dma-mapping.h>

#define HARDWARE_READ_LENGTH              ...
#define DMA_CONTROLLER_READ_CTRL          ...
#define DMA_CONTROLLER_READ_STAT          ...
#define DMA_CONTROLLER_READ_BUFFER_ADDR   ...
#define DMA_CONTROLLER_READ_BUFFER_LEN    ...

static struct device *dev;
static dma_addr_t *dma_handle;

ssize_t driver_read(struct file *instance, char __user *buffer,
        size_t max_bytes_to_read, loff_t *offset) {
    void *dma_hardware = ((struct instant_data*) instance->private_data)->hardware_ptr;
    void *dma_mem;
    size_t to_copy, not_copied;

    //Prüfe Hardwaremodul auf Beendigung seiner Verarbeitung (optional)
    if (!(*hardware & 0x02)) {
        return -EAGAIN;
    }

    // DMA-Speicher anfordern
    dma_mem = dma_alloc_coherent(dev, HARDWARE_READ_LENGTH, dma_handle, GFP_KERNEL);

    // DMA-Controller starten
    writeb(*dma_hardware + DMA_CONTROLLER_READ_CTRL, 1);
    // DMA-Controller Adresse des DMA-Speichers übergeben
    writel(*dma_hardware + DMA_CONTROLLER_READ_BUFFER_ADDR, dma_mem);
    // DMA-Controller Länge des DMA-Speichers übergeben
    writel(*dma_hardware + DMA_CONTROLLER_READ_BUFFER_LEN
            , HARDWARE_READ_LENGTH);

    // Beendigung der Übertragung des DMA-Controllers abwarten
    while (!((*dma_hardware + DMA_CONTROLLER_READ_STAT)&0x02));

    // Datentransport von DMA-Speicher zum Userspace
    to_copy = min(HARDWARE_READ_LENGTH, max_bytes_to_read);
    not_copied = copy_to_user(buffer, dma_mem, to_copy);

    // DMA-Speicher freigeben
    dma_free_coherent(dev, HARDWARE_READ_LENGTH, dma_mem, dma_handle);
```

```
    return to_copy - not_copied;
}
```

3.2.3 Applikation

Wie der entsprechende Treiber aufgebaut sein muss, wurde in den vorangegangenen Kapiteln ausführlich beschrieben. Dieser Abschnitt beschreibt die Anwendung, welche über einen Zeichen-Treiber mit dem Hardwaremodul kommuniziert. Es ist unabhängig der Technik (Memory Mapped I/O, Interrupt oder Direct Memory Access), wie das Modul mit dem Prozessorsystem verbunden ist, weil durch den Treiber eine einheitliche Schnittstelle zwischen Applikation und Hardware geschaffen wird. Die Applikation benutzt den Treiber, über die Systemcalls *open()*, *write()*, *read()* und *close(),* um mit dem Hardwaremodul zu kommunizieren. Um mit dem Modul zu kommunizieren, muss die Applikation den Treiber mit *open()* öffnen, die Daten mit *write()* übergeben, die Daten mittels *read()* entgegennehmen und den Treiber mit *close()* schließen.

3.3 Userspace-Treiber, Direktzugriff am Betriebssystem vorbei

Mit Hilfe eines Userspace-Treibers wird ein direkter Zugriff der Applikation auf die Register eines Hardwaremoduls erreicht. Des Weiteren kann ein Interrupt genutzt werden. Im Gegensatz zu Zeichen-Treibern wird der Adressbereich des Moduls in den virtuellen Adressbereich der Applikation abgebildet. Abbildung 3.4 zeigt schematisch wie der Zugriff auf das Hardwaremodul mit Userspace-Treiber zu realisieren ist.

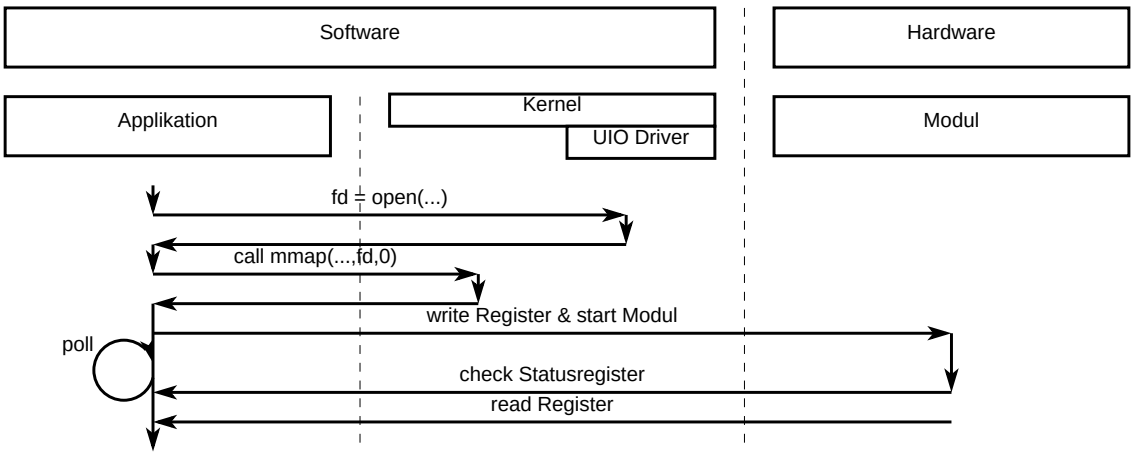

Abbildung 3.4: Schema: Applikation - Treiber(UIO) - Hardware

Für dieses Verfahren werden zwei Programme benötigt, ein Kernel-Treiber und die Applikation. Der Treiber übernimmt nicht die Funktion, mit dem Modul zu kommunizieren. Er ermöglicht, dass die Applikation auf den gewünschten Adressbereich zugreifen kann und wenn gewünscht auf Interrupts reagiert wird. Die Applikation kann über den eingeblendeten Adressraum des Hardwaremoduls mit diesem kommunizieren.

Mit wenigen Zeilen Quellcode kann gezeigt werden, wie ein solcher Kernel-Treiber aussieht. Der Quellcode des folgenden Beispiels stammt aus dem Artikel „Kernel- und Treiberprogrammierung mit dem Kernel 2.6 - Folge 36“ [8] des Linux Magazin. In diesem Artikel wurde die Funktionsweise eines Userspace-Treibers kurz dargestellt. Der Quellcode wurde angepasst, da kein logischer Speicherbereich in die Applikation abgebildet werden sollte, sondern ein physischer Adressbereich.

```
#include <linux/module.h>
#include <linux/platform_device.h>
#include <linux/uio_driver.h>
#include <linux/slab.h>
#include <asm/io.h>

struct uio_info kpart_info = {
    .name = "kpart",
    .version = "0.1",
    .irq = UIO_IRQ_NONE,
};

static int drv_kpart_probe(struct device *dev);
static int drv_kpart_remove(struct device *dev);

static struct resource *allin;

#define HARDWARE_ADDR           ...
#define HARDWARE_ADDR_RANGE     ...

static struct device_driver uio_dummy_driver = {
    .name = "kpart",
    .bus = &platform_bus_type,
    .probe = drv_kpart_probe,
    .remove = drv_kpart_remove,
};

static int drv_kpart_probe(struct device *dev) {
    printk("drv_kpart_probe(␣%p␣)n", dev);

    // Moduladresse eintragen
    kpart_info.mem[0].addr = HARDWARE_ADDR;
```

```
    if (kpart_info.mem[0].addr == 0)
        return -ENOMEM;
    // Speichertyp angeben
    kpart_info.mem[0].memtype = UIO_MEM_PHYS;
    // Größe des Speicherbereiches
    kpart_info.mem[0].size = HARDWARE_ADDR_RANGE;
    if (uio_register_device(dev, &kpart_info))
        return -ENODEV;
    return 0;
}

static int drv_kpart_remove(struct device *dev) {
    release_mem_region(HARDWARE_ADDR, HARDWARE_ADDR_RANGE);
    uio_unregister_device(&kpart_info);
    return 0;
}

static struct platform_device *uio_dummy_device;

static int __init uio_kpart_init(void) {
    uio_dummy_device = platform_device_register_simple("kpart", -1, NULL, 0);
    return driver_register(&uio_dummy_driver);
}

static void __exit uio_kpart_exit(void) {
    platform_device_unregister(uio_dummy_device);
    driver_unregister(&uio_dummy_driver);
}

module_init(uio_kpart_init);
module_exit(uio_kpart_exit);

MODULE_LICENSE("GPL");
```

3.3.1 Interrupt Service Routine

Das Nutzen von Interrupts bzw. ISR im Userspace-Treiber wird unterstützt. Für das Nutzen des Interrupts muss die Nummer des Interrupts als Variable *long irq* und die ISR als *irqreturn_t *handler()* des *struct uio_info* eingetragen werden. Damit die Applikation Interrupts nutzen kann, muss sie die Funktion *read()* oder *select()* des Userspace-Treibers aufrufen. Wenn kein Interrupt aufgetreten ist, wird der Prozess in der Funktion *read()* schlafen gelegt, sonst gibt read() die Anzahl der eingegangenen Interrupts zurück. Zu beachten ist, dass die Anzahl in einem 32-Bit-Datentyp zurückgegeben wird.

Es muss eine Funktion *read()* aufgerufen werden, welche diesen 32-Bit-Datentyp abruft. Das Lesen der 32 Bit wird in der Applikation aufgeführt.

Die Interrupt Service Routine überprüft, ob das Hardwaremodul den Interrupt ausgelöst hat. Ein Aufwecken der Funktion *read()* muss nicht ausgelöst werden, diese weckt auf, sobald die ISR mit IRQ_HANDLED quittiert wurde. Interrupt-Sharing kann bei Userspace-Treiber realisiert werden, indem die ISR mit IRQ_NONE quittiert, wenn der Interrupt nicht von dem zugehörigen Modul ausgelöst wurde. Nach Abarbeiten der ISR wird die nächste ISR aufgerufen, bei welcher der gleiche Interrupt verwendet wurde.

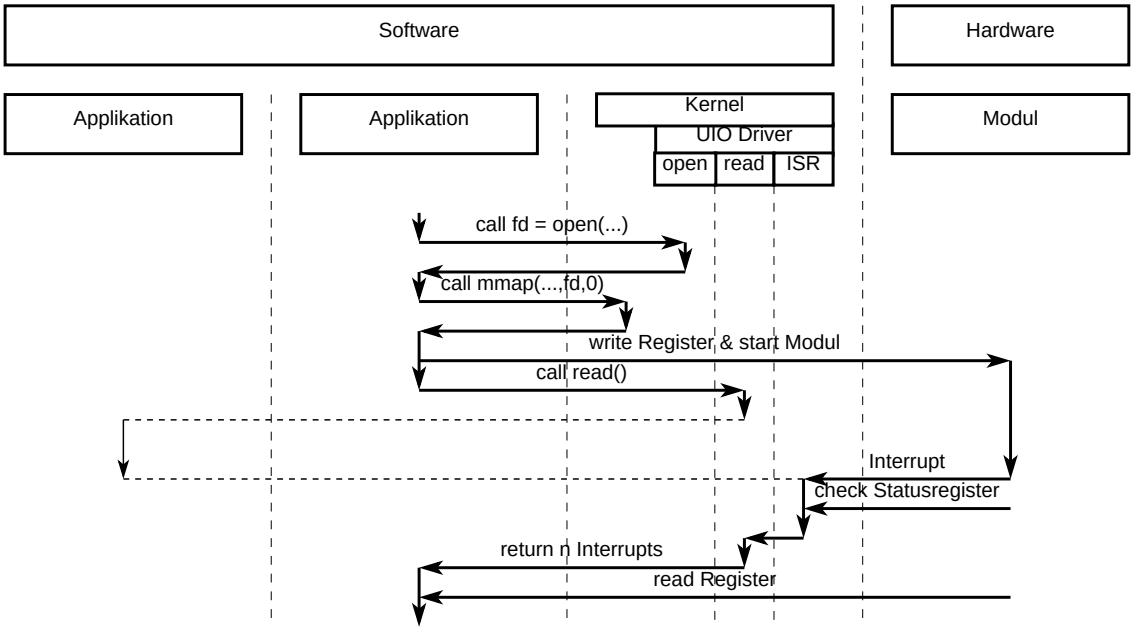

Abbildung 3.5: Schema: Applikation - Treiber(UIO/ISR) - Hardware

Abbildung 3.5 zeigt, wie die Kommunikation mittels Userspace-Treiber und Interrupt bzw. ISR aussieht. Im nachstehenden Listing wird die ISR dargestellt.

```
#define HARDWARE_ADDR    ...

static DECLARE_WAIT_QUEUE_HEAD(read_wait_irq);

irqreturn_t uio_isr(int irq, void *dev_id) {
    // prüfe Modul
    if (!readb(HARDWARE_ADDR) & 0x02) {
        // Interrupt wurde nicht durch Modul des Treibers ausgelöst
        return IRQ_NONE;
    }

    // Interrupt als bearbeitet gegenüber dem Kernel melden
    return IRQ_HANDLED;
}
```

3.3.2 Direct Memory Access

Das Nutzen von DMA durch den Userspace-Treiber wird nicht unterstützt. Es fehlt die Möglichkeit zusammenhängenden physischen Speicher anzufordern, weil das im Kernelspace zu realisieren ist. Dadurch kann während der Laufzeit kein DMA-Speicher reserviert werden. Es existieren Patches für die Anforderung von DMA-Speicher im Userspace-Treiber, mit dessen Hilfe ein Userspace-Treiber DMA unterstützt. Zum Beispiel gibt es ein Patch für den Kernel 2.6.28-rc6 von Edward Estabrook unter dem Namen Userspace I/O (UIO): Add support for userspace DMA [6].

3.3.3 Applikation

Im Gegensatz zur Applikation für den Zeichen-Treiber, muss die Applikation bei UIO die Steuerung des Moduls übernehmen. Benötigt wird der Zugriff der Applikation auf den Adressraum des Moduls. Die Funktion *mmap()* sorgt für die nötige Abbildung der physischen Hardwareadressen in den virtuellen Adressraum der Applikation. Bei Aufruf dieser Funktion wird der Filedeskriptor des Gerätetreibers benötigt, unter welcher der Userspace-Treiber eingehangen wurde. Die Userspace-Treiber werden unter „/dev/uioX“ geführt. Wichtig ist die Nummer, hier als „X“ dargestellt, welche die Applikation benötigt um den zugehörigen Treiber zu verwenden. Um herauszubekommen unter welcher Nummer der Treiber geführt wird, werden verschiedene Parameter des Treibers unter „/sys/class/uio/uioX/...“ bereit gestellt. Zum Beispiel könnte der Name und Version des Treibers überprüft werden, um den zugehörigen Treiber zu identifizieren.

```
#define DATA_LENGTH_TO_HW       ...
#define DATA_LENGTH_FROM_HW     ...
#define HARDWARE_WRITE          ...
#define HARDWARE_READ           ...
#define HARDWARE_ADDR_RANGE     ...

int use_hw(char dataToHw[], dataFromHw[]) {
    int fd;
    void *hardware;
    ssize_t nint;

    fd = open(UIO_DEV, O_RDONLY);
    if (fd < 0)
        return -1;

    hardware = mmap(NULL, HARDWARE_ADDR_RANGE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if (hardware == NULL) {
        close(fd);
```

```
        return EXIT_FAILURE;
    }

    memcpy(hardware + HARDWARE_WRITE, dataToHw, DATA_LENGTH_TO_HW);

    writeb(hardware, 0x01);

    if (read(fd, nint, sizeof (ssize_t)) < 0) {
        close(fd);
        return EXIT_FAILURE;
    }

    memcpy(dataFromHw, hardware + HARDWARE_READ, DATA_LENGTH_FROM_HW);

    return EXIT_SUCCESS;
}
```

4 Programmierbare Logik als Ressource

Bisher wurde eine Applikation unter einem Betriebssystem betrachtet. Da mehrere Applikationen unter dem Betriebssystem ausgeführt werden können und evtl. mehrere Applikationen von der programmierbaren Logik profitieren wollen, wird noch ein Konzept benötigt, bei welchem sich mehrere Applikationen die programmierbare Logik teilen. Die Ressource, PL, kann auf verschiedene Arten für die Applikationen aufgeteilt werden. Die Ressource kann zeitlich sowie physisch geteilt zur Verfügung stehen. Zeitlich teilen soll bedeuten, dass die Applikation die gesamte Ressource für eine definierte Zeit zur Verfügung gestellt bekommt. Das physische Teilen bedeutet die PL wird in Bereiche aufgeteilt, welche den einzelnen Applikationen zur Verfügung gestellt werden. Beide Verfahren unterscheiden sich deutlich voneinander und ihre Vor- sowie Nachteile werden in den folgenden Abschnitten beschrieben.

Beim zeitlichen Aufteilen der programmierbaren Logik ist der große Vorteil, dass die Anwendung den gesamten Bereich zur Verfügung gestellt bekommt. Somit wird das Modul nur durch die Größe der programmierbaren Logik begrenzt und es lassen sich größere Schaltungen unterbringen. Daraus resultieren folgende Nachteile:

- Ungenutzte Bereiche bei kleinen Schaltungen in der PL
- Blockieren der gesamte Ressource durch ein Hardwaremodul

Es könnte ein Hardwaremodul während seiner Verarbeitung durch ein anderes Hardwaremodul ersetzt werden. Die Folge der Ersetzung wäre, dass der Verarbeitungsprozess des Moduls unterbrochen wird. Bedingt durch die Unterbrechung müsste die Verarbeitung entweder per Software erfolgen oder zu einem späteren Zeitpunkt im Hardwaremodul von vorn beginnen. Die Aufgabe des Hardwaremoduls sollte sein, die Verarbeitung zu beschleunigen. Wegen diese Aufgabe sollte eine Unterbrechung verhindert werden.

Bei der physischen Aufteilung der programmierbaren Logik werden die Nachteile der zeitlichen Aufteilung verhindert. Zum Beispiel können mehrere Hardwaremodule ihre Arbeit parallel verrichten. Durch das Aufteilen können Module in der programmierbaren Logik erhalten bleiben und es wird verhindert, dass die PL mit Zeitaufwand neu konfiguriert werden muss. Der maßgebliche Vorteil der zeitlichen Aufteilung wird zum Nachteil der physischen Aufteilung. Eine Schaltung bei physischer Auteilung wird durch die programmierbaren Bereiche begrenzt. Bei der zeitlichen Aufteilung konnte die ge-

samte programmierbare Logik verwendet werden. Um diesen Nachteil zu verringern, wäre es ein Ansatz mehrere Bereiche im Verbund zu nutzen. In Tabelle 4.1 werden die Vor- und Nachteile der Verfahren nochmals kurz gegenübergestellt.

zeitliche Aufteilung	physische Aufteilung
nur ein Modul in einem Zeitraum	zeitlich parallel arbeitende Module
große Schaltungen realisierbar für komplexere Algorithmen	Schaltung in ihrer Größe begrenzt
ungenutzte Bereiche der programmierbaren Logik bei kleinen Schaltungen	bessere Ausnutzung der programmierbaren Logik bei kleineren Schaltungen möglich
ständige Konfiguration der programmierbaren Logik bei Modulwechsel benötigt Zeit	Module müssen in programmierbarer Logik nicht ständig ausgetauscht werden

Tabelle 4.1: Vergleich der Aufteilung von programmierbarer Logik

Die voranstehende Vorbetrachtung stellt klar, welche Kompromisse man jeweils eingehen müsste. Dem Aspekt des blockierenden Moduls bei zeitlicher Aufteilung sollte eine große Bedeutung zugeteilt werden. Weil das Betriebssystem keinen Einfluss auf die Dauer einer Verarbeitung im Modul hat, könnte ein konfiguriertes Modul die PL unbestimmt blockieren oder müsste unterbrochen werden. Ein Scheduling könnte funktionieren, wenn der Zustand des Moduls gespeichert und wiederhergestellt werden könnte. Durch die Vielfalt der Schaltungsmöglichkeiten ist dies schwer realisierbar. Aufgrund dieses Problems wird nachfolgend nur die physische Aufteilung mit der partiellen Rekonfiguration betrachtet. Bei physischer Aufteilung stehen weiterhin programmierbare Bereiche zu Verfügung, wenn ein rekonfigurierbarer Bereich auf unbestimmte Zeit blockiert ist.

4.1 Partielle Rekonfiguration

In Abbildung 1.2 der Einleitung wurde veranschaulicht, wie verschiedene Applikationen auf unterschiedliche Module zugreifen. Im vorangehenden Abschnitt wurde dieses Modell mit der physischen Aufteilung genauer erfasst. Prinzipiell wird zu Beginn der Inbetriebnahme des Systems die PL konfiguriert. Damit wird die interne Verschaltung der PL vorgenommen und gewöhnlich während der Laufzeit nicht verändert. Es sei denn, die Logik im System muss aktualisiert werden, obwohl das System nicht heruntergefahren werden kann. Mit partieller Rekonfiguration wird zusätzlich das Verändern eines Bereiches der programmierbaren Logik zur Laufzeit ermöglicht. Auf das Verfahren, wie partielle Hardwaremodule generiert werden, wird an dieser Stelle nicht eingegangen.

Eine Anleitung für partielle Rekonfiguration gibt es von Xilinx „Zynq 7000 Partial Reconfiguration Reference Design“ [12], auf welche später Bezug genommen wird.

Dieses Verfahren ermöglicht, dass ein Bereich der PL für Userspace-Programme zur Verfügung gestellt werden kann. Somit wird die PL als Ressource betrachtet, vergleichbar wie der Speicher. Um diese Ressource sinnvoll zu nutzen, wird der Zugriff zum Programmieren der Logik, der Zugriff auf das Hardwaremodul sowie eine Verwaltung der rekonfigurierbaren Bereiche nötig. Wie auf das Hardwaremodul zugegriffen werden kann, wurde in den Kapiteln 2 und 3 einschlägig demonstriert. Ungeachtet dessen sind für partielle Rekonfiguration Besonderheiten zu beachten. Beispielsweise wird eine Adressbindung bei Generierung der Hardwaremodule mittels HLS und Vivado am *AXI bus interface* vorgenommen. Diese Adressbindung könnte zum Problem werden. Unabhängige Applikationen könnten Hardwaremodule mit gleichen Adressen zur selben Zeit in der programmierbaren Logik einsetzen wollen, was zwangsläufig zu Problemen führt. Abhilfe kann durch verschiedene Ansätze geschaffen werden.

Ein Ansatz ist eine Schnittstelle zu integrieren, welche nicht auf Adressen basiert. Vorgestellt wurde mit dem DMA-Controller das *AXI-Stream Bus Interface*. Wird jedem rekonfigurierbaren Bereich ein DMA-Controller zur Verfügung gestellt, können Module ohne Adressen in den Bereichen konfiguriert werden. Der DMA-Controller liegt außerhalb des rekonfigurierbaren Bereiches und erhält eine feste Adresse. Wenn der Applikation ein Bereich für sein Modul zur Verfügung gestellt wird, muss der Applikation ein direkter oder indirekter Zugriff auf den DMA-Controllers ermöglicht werden.

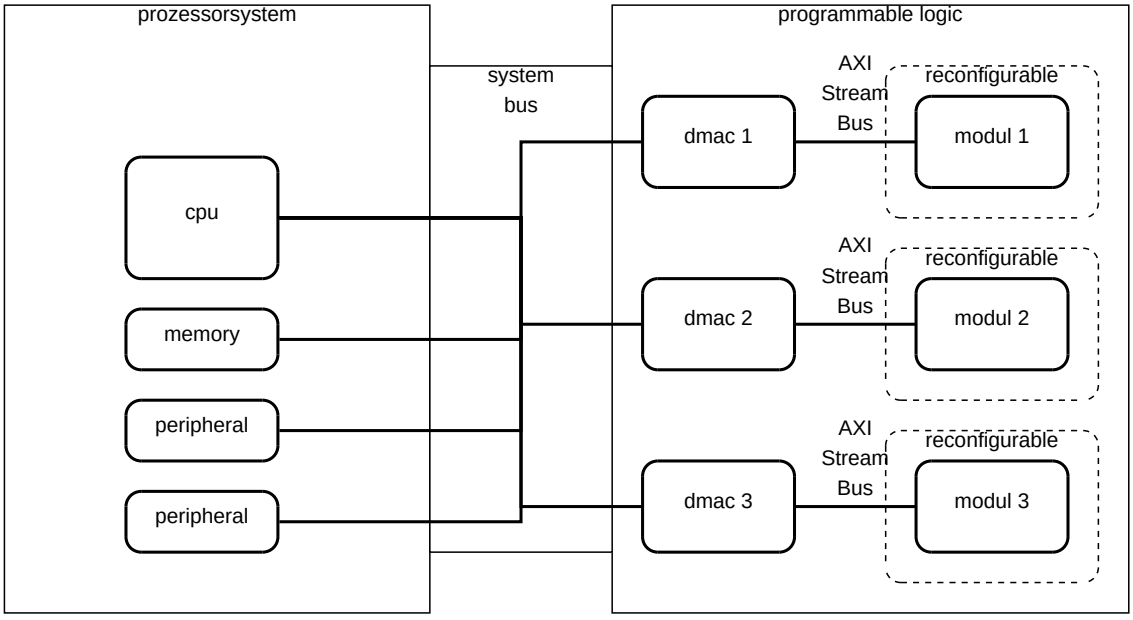

Abbildung 4.1: partielle Rekonfiguration - Direct Memory Access

Eine andere Möglichkeit besteht darin, eine Adresse über eine Schnittstelle für den rekonfigurierbaren Bereich zur Verfügung zu stellen. Wenn ein Modul unter einer Adresse am Systembus zur Verfügung stehen soll, muss es die zur Verfügung gestellte Adresse für die Adressbindung zur Laufzeit verwenden. Dem Treiber wird daraufhin die Adresse des rekonfigurierbaren Bereiches zur Verfügung gestellt. Diese Möglichkeit besteht zurzeit nicht bei den eingesetzten Tools und müsste evtl. auf der unterster Ebene in *VHDL* oder *Verilog* umgesetzt werden.

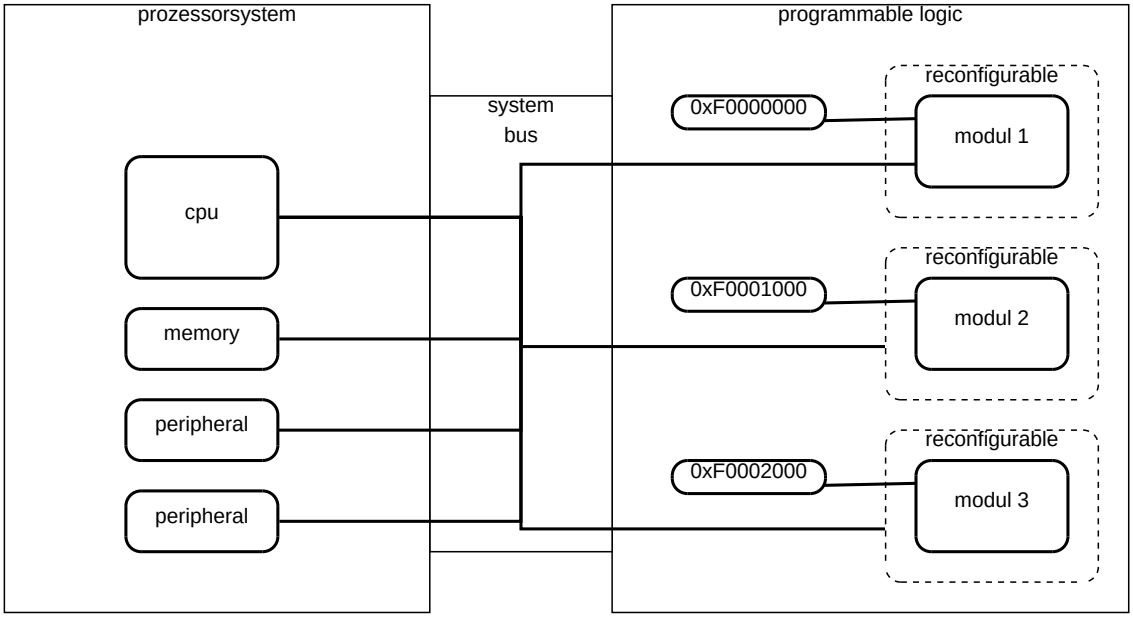

Abbildung 4.2: partielle Rekonfiguration- variable Adressbindung

Zugriff auf die PL unter Linux zur Rekonfiguration erhält man über die Gerätedatei */dev/xdevcfg*. Verschiedene Patches werden für den Linux Kernel benötigt. Der eingangs erwähnten Anleitung [12] kann unter Punkt „5.2 Building the Linux Kernel Image and Device Tree Blob“ entnommen werden, welche Patches benötigt werden und wie diese eingespielt werden. Um partielle Rekonfiguration nutzen zu können, muss

```
$ echo 1 > /sys/dev/amba.0/f8007000.devcfg/is_partial_bitstream
```

vor der partielle Rekonfiguration aus der Anleitung [XAPP 1159] ausgeführt werden. Wenn ein Bereich der programmierbaren Logik konfiguriert werden soll, muss die Applikation einen Bitstream zur Verfügung stellen, welcher für diesen Bereich generiert wurde. Eine Anwendung müsste *n* Bitstreams bereithalten, um *n* rekonfigurierbare Bereiche abzudecken. Dies ist notwendig, da vom Hersteller Xilinx nicht bekannt gegeben wird, wie die PL auf unterster Schaltungsebene aufgebaut ist und ob die PL Symmetrien aufweist. In der Arbeit „Multiple-Clone Configuration of Relocatable Partial Bitstreams in Xilinx Virtex“ [1] wurden solche Symmetrien auf einem Xilinx Virtex nachgewiesen. Sol-

che symmetrischen Bereiche müssen auf dem Zynq nachgewiesen werden, damit die Anwendung nicht für alle rekonfigurierbaren Bereiche Bitstreams bereithalten muss.

4.2 Ressourcenverwaltung und Zugriffsschutz

Für die Ressourcenverwaltung werden weitere Funktionalitäten benötigt, welche die rekonfigurierbaren Bereiche verwalten. Die folgenden Funktionalitäten sollten implementiert werden:

- Reservieren eines konfigurierbaren Bereiches
- Freigeben des vorher reservierten konfigurierbaren Bereiches
- Konfigurieren eines Moduls im konfigurierbaren Bereich
- Lese- und Schreibzugriff auf das konfigurierte Modul

Wichtig bei der Implementierung ist ein Zugriffsschutz, welcher dem der Speicherverwaltung entsprechen sollte. Zwei Szenarien des Zugriffsschutzes sind abzudecken, erstens das Konfigurieren der Bereiche und zweitens der Zugriff auf die konfigurierten Module.

1. Beim Konfigurieren der Bereiche, darf eine Applikation keinen Bereich konfigurieren können, welcher der Applikation nicht zugeordnet wurde. Der von der Applikation bereitgestellte Bitstream enthält die Information, wie die programmierbare Logik konfiguriert wird. Schlussfolgerung daraus ist, dass die Konfiguration der programmierbaren Logik über die Applikation definiert wird. Die Funktion zum Konfigurieren müsste den Bitstream verifizieren, ob dieser für den Bereich ist, welcher von der Applikation reserviert, wurde. Sollte sich ein Verfahren auf der Arbeit „Multiple-Clone Configuration of Relocatable Partial Bitstreams in Xilinx Virtex“ [1] für die SoCs Zynq-7000 umsetzen lassen, muss die Funktion zum Konfigurieren das Anpassen des Bitstreams für den reservierten Bereich vornehmen. In welchem Ausmaß eine Verifizierung nötig wird, ist zu überprüfen.
2. Szenario zwei des Zugriffsschutzes, ist der Zugriff auf das Hardwaremodul. Die im Abschnitt 3.2 vorgestellten Treiberkonzepte reichen hier nicht. Eine Anwendung kann bisher jeden Treiber für die Module öffnen und würde uneingeschränkten Zugriff auf das Modul besitzen. Dieser uneingeschränkte Zugriff muss durch einen exklusiven Zugriff zwischen Applikation und Hardwaremodul verhindert werden.

Um diese zwei Szenarien zu realisieren, ist es wichtig, dass die Ressourcenverwaltung mit Zugriffsschutz im Kernelspace erfolgt. Wenn die Ressourcenverwaltung mit Zugriffs-

schutz über Bibliotheken im Userspace realisiert würde, kann der Zugriffsschutz umgangen werden. Die Ressourcenverwaltung im Kernelspace könnte auf mehrere Möglichkeiten realisiert werden. Zwei dieser Möglichkeiten sind naheliegend:

- Systemcalls, wie bei der Speicherverwaltung mit den Systemcalls *alloc()* oder *free()*
- Treiber, welcher über standardisierten Systemcalls *open()*, *write()*, *read()*, *close()*, sowie *ioctl()* zur Verfügung steht

Die Variante über eigene Systemcalls wäre die elegantere und schwierigere Lösung. Eleganter dahingehend, dass für jede Funktion ein gesonderter Systemcall zur Verfügung steht, was für die Programmierung einfacher zu Verstehen ist. Schwieriger, weil das einen größeren Eingriff in den Kernel darstellt im Vergleich zu einem Treiber. Um die Einfachheit der Systemcalls bei Treibern zu erreichen, könnte eine Abstraktion mittels Bibliotheken vorgenommen werden. Mit dieser Bibliothek könnten Befehle zur Verfügung stehen, welche eine Syntax analog zu *alloc()* oder *free()* aufweisen und einen Treiber benutzen, um auf die Ressourcen zuzugreifen. Des Weiteren erscheint eine Implementierung von Systemcalls für verschiedene Prozessorsysteme nicht sinnvoll, weil programmierbare Logik nicht bei allen Systemen integriert ist, sondern einzig bei einer kleinen Auswahl. Aus den vorangegangenen Gründen wird im Folgenden auf die Implementierung des Treibers, nicht die über Systemcalls, für die gewünschten Funktionalitäten eingegangen.

Bisher wurde davon ausgegangen, dass ein Treiber ein bestimmtes Modul bedient. Das heißt, jedes Modul besitzt seinen eigenen Treiber. Wird stattdessen ein Treiber für alle Module benutzt, kann ein exklusiver Zugriff durch den Treiber realisiert werden. Da in Kapitel 3 die Treiber-Funktionen für die Hardwaremodule beschrieben wurde, können diese Funktionen wiederverwendet werden. Sie müssen für Ressourcenverwaltung und Zugriffsschutz erweitert werden. Die Abbildung 4.3 und 4.4 zeigen wie Zeichen-Treiber und Userspace-Treiber umgesetzt werden könnten.

Ein Problem beim Userspace-Treiber ergibt sich beim Konfigurieren der Module, weil für den Userspace-Treiber kein Zugriff auf Hardware vorgesehen ist. Aus diesem Grund müsste der Userspace-Treiber in einem Treiber gekapselt werden, welcher ioctl() aufweist. Eine Implementation würde Ähnlichkeiten zum Zeichen-Treiber aufweisen und wird an dieser Stelle vernachlässigt.

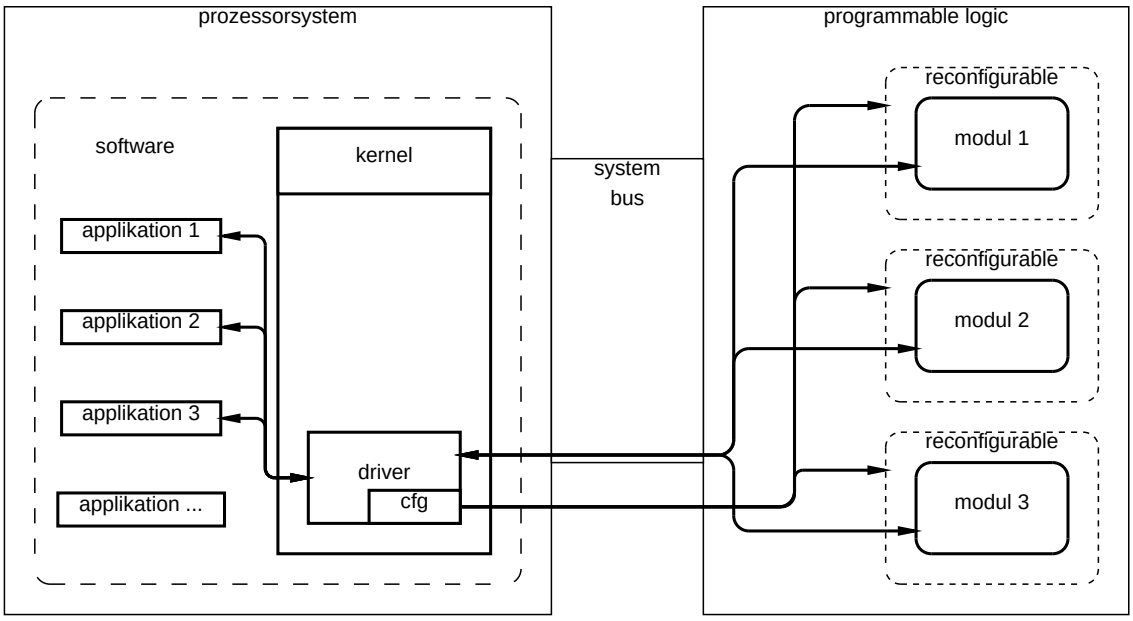

Abbildung 4.3: Zeichen-Treiber mit Verwaltung

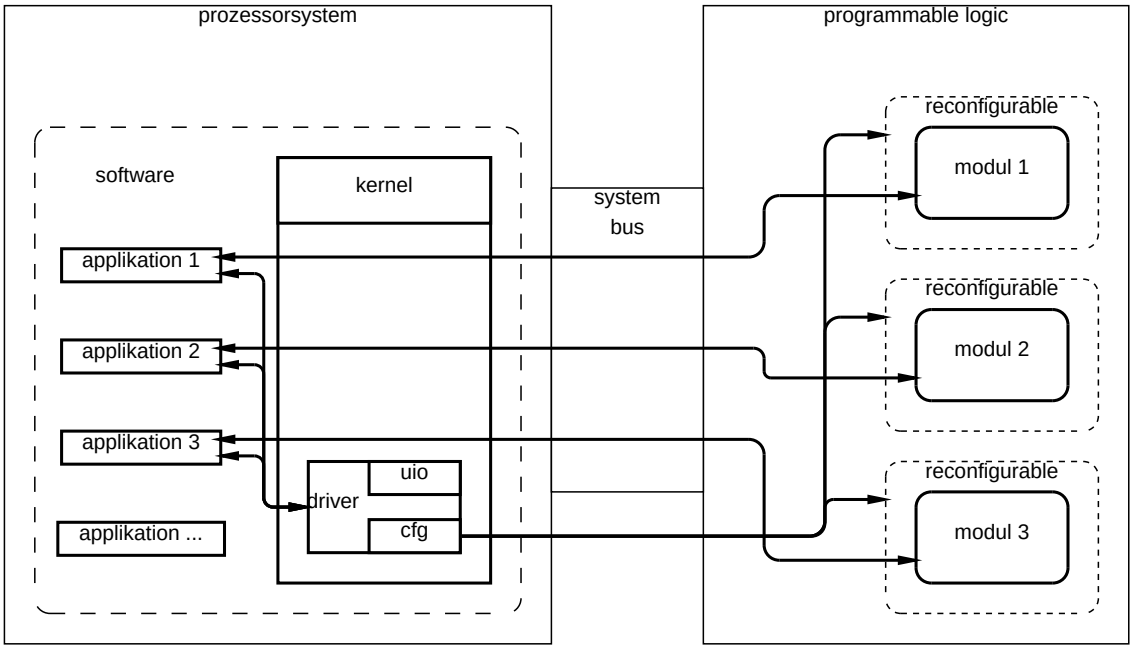

Abbildung 4.4: Userspace-Treiber mit Verwaltung

Um die Ressourcen verwalten zu können, muss festgehalten werden, wie viele dieser Ressourcen zur Verfügung stehen. Für die freien Ressourcen würden sich verkettete Listen anbieten. Die Einträge der Liste enthalten Referenzen auf die Ressourcen. Um ständiges Reservieren und Freigeben von Speicher für die Einträge zu verhindern, werden diese fest in einem Array hinterlegt. In Abbildung 4.5 wird die verkettete Liste dargestellt.

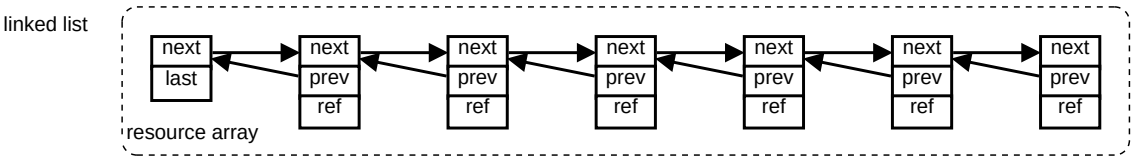

Abbildung 4.5: verkettete Liste

Bei einem *open()* wird geprüft, ob noch Ressourcen zur Verfügung stehen, indem festgestellt wird, ob die Liste leer ist. Im Fall, dass kein Eintrag in der Liste vorhanden ist, bricht *open()* mit einer Fehlermeldung ab, z.B. *EAGAIN*. Die *open()* aufrufende Applikation kann am Fehlercode feststellen, dass zurzeit keine Ressource zur Verfügung steht. Wenn ein Eintrag in der Liste ist, wird der erste Eintrag referenziert und aus der Liste entfernt. Wenn die Ressource nicht mehr benötigt wird, ist per *close()* der Eintrag mit der Referenz in die Liste hinzuzufügen. Damit kann erkannt werden, dass die Ressource zur Verfügung steht. Der Einfachheit wegen wird hier das Reservieren einer Ressource in Betracht gezogen. Um die Referenzen der Ressource für spätere Aufgaben verwenden zu können, muss diese an einer Stelle hinterlegt werden. In Abschnitt 3.2 wurde ein *struct instant_data* verwendet, um Daten für eine Treiberinstanz zu hinterlegen. Dieses *struct instant_data* müsste um einen Pointer auf einen Listeneintrag erweitert werden, welcher die Ressourcenreferenz enthält. Abbildung 4.6 stellt dar, wie die Treiberinstanzen auf einen Listeneintrag verweisen, welcher aus der Liste der freien Ressourcen entfernt wurde.

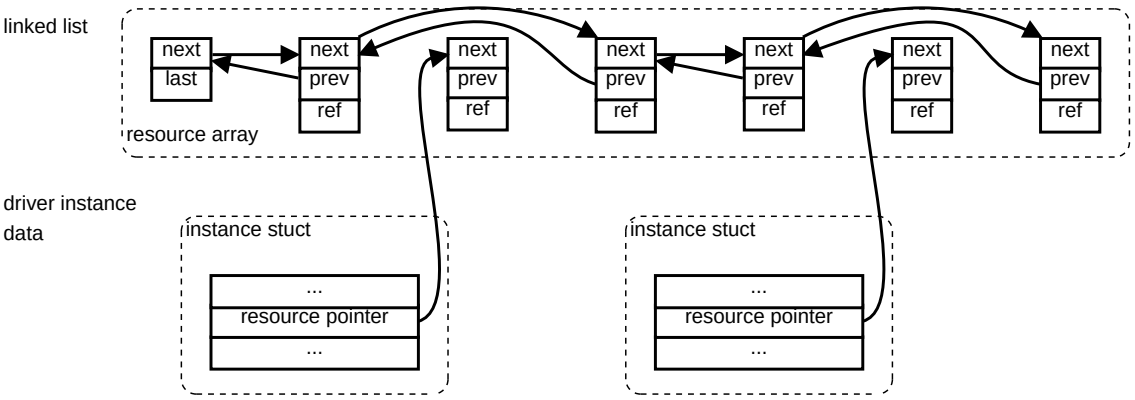

Abbildung 4.6: verkettete Liste und Treiberinstanz

Weiterhin muss der Adressbereich der Hardwaremodule reserviert werden. Nicht relevant ist, ob über einen DMA-Controller oder über Register auf das Modul zugegriffen wird, wenn eine der Techniken aus Abschnitt 4.1 angewendet wurde. Es macht keinen Unterschied, weil die Adresse des Moduls oder des DMA-Controllers dem rekonfigurierbaren Bereiche zugeordnet ist. Das bedeutet, dass dem rekonfigurierbaren Bereich der entsprechende Adressbereich für die Treiberinstanz reserviert werden muss. Wenn man davon ausgeht, dass die Ressourcenreferenzen einer Durchnummerierung 0..n

entspricht und die Adressbereiche der Module bzw. DMA-Controller hintereinander liegen, sowie gleich groß sind, kann für *request_mem_region()* und *ioremap()* die Adresse des Moduls folgend berechnet werden:

$$Modulzugriffsadresse = Basis + Adressbereich * Referenz$$

Durch das Hinterlegen der Adresse für den Zugriff auf ein Hardwaremodul im *struct instant_data* einer Treiberinstanz in der Funktion *open()* können die Funktionen *write()* sowie *read()* verwendet werden, wie in Kapitel 3.2 aufgezeigt wurde.

Offen bleibt das Konfigurieren der Bereich für die Hardwaremodule. Für das Konfigurieren könnte die Funktion *ioctl()* im Treiber implementiert werden. Da weder das Verifizieren, noch das Verändern eines Bitstreams genauer untersucht wurde, konnte bisher keine Implementierung vorgenommen werden. Das Konfigurieren der PL zum Beispiel im xdevcfg-Treiber von Xilinx wurde nicht genauer untersucht, um das Konfigurieren im eigenen Treiber zu implementieren. Deshalb wird in der Treiberimplentierung ein Rumpf für die Funktion der Verifizierung bzw. Veränderung und Konfigurierung des Bitstreams aufgezeigt.

5 Fazit und Ausblick

Das Grundproblem dieser Arbeit „Wie kann programmierbare Logik zur Unterstützung von Userspace-Programmen realisiert werden?“ wurde in den vorangegangenen Kapiteln ausführlich erläutert. Einige Details wurden bei dieser Arbeit wenig untersucht und lassen Raum für weitere Arbeiten.

5.1 Fazit

In den vorhergehenden Kapiteln wurde gezeigt, wie Hardwaremodule über die verschiedenen Schnittstellen an ein Prozessorsystem angebunden und ein Zugriff auf diese Hardwaremodule über Software realisiert werden können. In Kapitel 2 wurden die Grundlagen für die Hardwareanbindung geschaffen. Vorgestellt wurden die Schnittstellen Bus, Stream und Interrupt, mit den Techniken Memory Mapped I/O, Interrupt und Direct Memory Access. Damit programmierbare Logik durch Applikationen genutzt werden kann, wurden zu den Grundlagen der Hardware auch die Grundlagen der Software im dritten Kapitel aufgezeigt. Diese zwei Teile stellen heraus, welche der überprüften Techniken für Anbindung und Zugriff der Software zum Hardwaremodul geeignet sind. Es kann keine Aussage darüber getroffen werden, welche Technik zu favorisieren ist. Es kommt auf den Anwendungsfall an, welche der Techniken sich optimal auswirkt. Richtungsweisende Faktoren sind die Datenmenge der Übertragung und die Verarbeitungszeit des Moduls.

Ein Beispiel für den Faktor Datenmenge ist die Übertragung von wenigen Bytes, weil die Übertragung von 9 Byte der Größe entspricht, welche der eingesetzte DMA-Controller als Parameter benötigt. Aus diesem Zusammenhang und dem das die Datenübertragung zum Modul oder DMA-Controller mit der gleichen Geschwindigkeit über den selben Bus erfolgt, ergibt sich ein Vorteil der direkten Übertragung von wenigen Bytes ohne DMA. Die optimale Wahl des Treibers kann ebenfalls von den Faktoren der Datenmenge und Verarbeitungszeit abhängen. Ein Userspace-Treiber hat den Vorteil, dass die Abfrage eines Ergebnisses sofort nach dem Start der Verarbeitung in einem Modul realisierbar ist, wenn die Applikation nicht unterbrochen wird. Ein Zeichentreiber mit DMA und Interrupt würde seine Vorteile optimal nutzen können, wenn eine große Menge an Daten zu übertragen ist und die Verarbeitungszeit es zulässt, dass der Prozessor andere Prozesse ausführen könne.

Im zweiten Teil, Kapitel 4, wurde auf die programmierbare Logik eingegangen. Das Kapitel beschreibt, wie die programmierbare Logik als Ressource zu verwenden ist. Es wird gezeigt, wie die Ressource für Applikationen aufgeteilt werden kann, und wie eine Verwaltung der Ressource zu realisieren ist. Eingegangen wird auf die gegenseitige Einflussnahme von Applikationen untereinander und wie mit einem Zugriffsschutz entgegengewirkt werden kann. Beim Zugriff auf die Module kann durch den Treiber ein ausreichender Schutz gegen Fremdzugriff realisiert werden. Untersucht werden muss noch das Verifizieren bzw. das Anpassen eines Bitstreams, um einen Schutz gegen unautorisiertes Konfigurieren zu erhalten.

Bei der Aufteilung der Ressource wurden zwei Ansatzpunkte aufgezeigt, welche Nachteile aufwiesen. Der Nachteil der blockierten Ressource stellt ein Problem für die zeitliche sowie physische Aufteilung der Ressource dar. Bei der physischen Aufteilung spielt er eine kleinere Rolle, im Gegensatz zur zeitlichen Aufteilung, weil ein blockieren nicht die gesamte Ressource betrifft. Mit der physischen Aufteilung geht der Nachteil der Begrenzung der Programmierbaren Logik einher.

5.2 Ausblick

In dieser Arbeit wurde eine Basis für die Entwicklung von Unterstützung der Userspace-Programme durch programmierbare Logik geschaffen. Diese Basis zeigt die Schwäche der Konfiguration bzw. der unautorisierten Konfiguration auf. Diese Schwäche sollte primäres Ziel für weitere Arbeiten sein, weil sie eine Schwachstelle für die gesamte Systemstabilität darstellt. Wichtig für weitere Arbeiten sind die Probleme, welche durch die Ressource bzw. Ressourcenaufteilung entstehen. Zu untersuchen ist, wie programmierbare Bereiche der Logik zusammenarbeiten können, oder ob Hardwaremodule mit verdrängendem Scheduling realisierbar sind.

Der letzte Ansatz für weitere Untersuchungen ist der optimale Einsatz von Treibern. Für die Wahl des Treibers müsste untersucht werden, wie Datenübertragung und Verarbeitung zu realisieren sind, damit sich ein minimaler Zeit- und Ressourcenaufwand ergibt. Bei Ressourcenaufwand ist die Ressource programmierbare Logik sowie der Prozessor gemeint. Der Ressourcenaufwand des Prozessors sollte im Vergleich Hardwaremodul und Softwarelösung eines Algorithmus untersucht werden. Wird der Prozessor beim Einsatz eines Hardwaremoduls nicht entlastet, würde sich die Umsetzung nicht lohnen. Aus diesem Zusammenhang ist zu untersuchen, welche Aufgaben für eine Umsetzung als Hardwaremodul geeignet sind.

Literaturverzeichnis

[1] ALI EBRAHIM, Xabier Iturbe Chuan H. Khaled Benkrid B. Khaled Benkrid: Multiple-Clone Configuration of Relocatable Partial Bitstreams in Xilinx VirtexFPGAs. In: *Linux Magazin* (2013), April

[2] ARM LTD. (Hrsg.): *AMBA®AXI™and ACE™Protocol Specification*. D. ARM Ltd., October 2013

[3] AVNET ELECTRONICS MARKETING (Hrsg.): *ZedBoard Brochure English Version*. Avnet Electronics Marketing, September 2012

[4] BEIERLEIN, T. ; HAGENBRUCH, O.: *Taschenbuch Mikroprozessortechnik*. Fachbuchverl. Leipzig im Carl-Hanser-Verlag, 2011. – ISBN 9783446220720

[5] CORBET, Jonathan ; RUBINI, Alessandro ; KROAH-HARTMAN, Greg: *Linux device drivers - where the Kernel meets the hardware (3. ed.)*. O'Reilly, 2005. – I–XVII, 1–615 S. – ISBN 978–0–596–00590–0

[6] ESTABROOK, Edward: *Userspace I/O (UIO): Add support for userspace DMA@ONLINE*. http://lwn.net/Articles/309529/. – Abrufdatum: 26.07.2013

[7] ONLINE heise: *heise online@ONLINE*. http://www.heise.de/newsticker/meldung/Xilinx-kauft-Triscend-95015.html. – Abrufdatum: 09.09.2013

[8] QUADE, Jürgen ; KUNST, Eva-Katharina: Kernel- und Treiberprogrammierung mit dem Kernel 2.6 - Folge 36. In: *Linux Magazin* (2007)

[9] QUADE, Jürgen ; KUNST, Eva-Katharina: *Linux-Treiber entwickeln - Gerätetreiber für Kernel 2.6 systematisch eingeführt (3. Aufl.)*. dpunkt.verlag, 2011. – ISBN 978–3–89864–696–3

[10] RINGSLEBEN, Frederic: *Praktikumsbericht: High Level Synthesis*. Mai 2013

[11] WOLF, W.H.: *Computers and Components: Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers, 2001 (The Morgan Kaufmann Series in Computer Architecture and Design Series). – ISBN 9781558605411

[12] XILINX, Inc: *Zynq 7000 Partial Reconfiguration Reference Design@ONLINE*. http://www.wiki.xilinx.com/Zynq+7000+Partial+Reconfiguration+Reference+Design. – Abrufdatum: 29.08.2013

[13] XILINX, INC (Hrsg.): *LogiCORE IP AXI DMA v7.0 (PG021)*. 7.0. Xilinx, Inc, March 2013

[14] XILINX, INC (Hrsg.): *Xilinx Vivado Design Suite User Guide: High-Level Synthesis (UG902)*. 2013.2. Xilinx, Inc, June 2013

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 18. September 2013